Tampere University

Matti Vaattovaara

PERFORMANCE OF MODEL-BASED TESTING FOR AN ANDROID APPLICATION

Faculty of Information Technology and Communication Sciences
Master's Thesis
October 2019

ABSTRACT

Matti Vaattovaara: Performance of Model-Based Testing for an Android Application
Master's Thesis
Tampere University
Master of Science in Technology
October 2019

The supply of Android applications is large, and the market is highly competitive. Bugs and performance issues in an application are reasons for a user to switch to a more stable competitor. Automated graphical user interface (GUI) testing of mobile applications is one of the ways to improve their quality. There are several techniques to implement GUI testing, each of which has its strengths and weaknesses.

In this study, a model-based GUI test automation was implemented for an Android application. The performance of model-based test automation was evaluated based on fault detection and coverage. This was done by using the test automation on an Android application under development over three months. As a point of reference, the same versions of the application were tested by a fairly sophisticated manually scripted UI test automation over the same period. To mitigate the effect of cost, the amount of effort put to implementing each of the solutions was evened out. In addition to fault detection and coverage, the characteristics of the implementations were compared in terms of applicability and cost-effectiveness over the development cycle.

The results of the study were in line with the general theory of model-based testing (MBT). It was concluded that model-based testing achieved better use case coverage and fault detection than the manually scripted test automation, provided that the effort was big enough for the advantages of model-based testing paradigm to manifest. Model-based testing provided the ability to run noticeably longer test cases with hardly any extra effort, thus allowing testing for memory leak issues. Another model-based advantage was the ability to focus or avoid testing on specified areas of the system under test (SUT) using strategies. On the other hand, the manually scripted test automation solution was easier to implement, faster to start yielding benefits and more flexible in terms of platform and device.

Keywords: model-based, test automation, coverage, fault detection

TIIVISTELMÄ

Matti Vaattovaara: Mallipohjaisen testauksen suorituskyky android-sovelluksessa
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-tutkinto-ohjelma
Lokakuu 2019

Android-sovellusten tarjonta on suuri ja markkinoilla on kova kilpailu. Sovelluksen virheet ja suorituskykyongelmat ovat syitä käyttäjälle siirtyä vakaampaan kilpailijaan. Mobiilisovellusten graafisen käyttöliittymätestauksen testiautomaatio on yksi tapa parantaa niiden laatua. Käyttöliittymän testiautomaation luontiin on useita tekniikoita, joista jokaisella on vahvuuksia ja heikkouksia.

Tässä tutkimuksessa toteutettiin mallipohjainen käyttöliittymän testiautomaatioratkaisu Android-sovellukselle. Mallipohjaisen testauksen suoriutumista arvioitiin havaittujen vikojen ja testien kattavuuden pohjalta. Tämä tehtiin käyttämällä testiautomaatioratkaisua kehitteillä olevan Android-sovelluksen testaukseen. Vertailukohtana samaa sovellusta testattiin samoissa kehitysvaiheissa myös perinteisellä, manuaalisesti komentosarjoiksi kirjoitetulla testiautomaatioratkaisulla. Työpanoksen merkitys pyrittiin minimoimaan käyttämällä kunkin testiautomaatioratkaisun kehitykseen jokseenkin sama määrä työtä. Vikojen havainnoinnin ja kattavuuden lisäksi toteutuksia vertailtiin sovellettavuuden ja kustannustehokkuuden eri kehitysvaiheissa kannalta.

Tutkimuksen tulokset olivat hyvin pitkälti linjassa yleisen mallipohjaisen testaukseen liittyvän teoria kanssa. Mallipohjaisen testauksen todettiin saavuttavan paremman kattavuuden käyttötapausten testauksessa ja vikojen havaitsemisessa verrattuna komentosarjoina kirjoitettuun testiautomaatioon, sillä oletuksella että testiautomaatiota kehitettiin tarpeeksi pitkälle, jotta mallipohjaisuuden hyödyt alkoivat ilmetä. Mallipohjainen testaus mahdollisti selkeästi pidempien testitapausten ajon lähes olemattomalla lisäpanoksella ja siten testauksen, joka voi havaita muistivuotoongelmia. Toinen mallipohjaisuuteen liittyvä etu oli kyky keskittää tai välttää testausta strategioiden avulla tietyillä testattavan sovelluksen alueilla. Toisaalta komentosarjoiksi kirjoitettu testiautomaatio oli helpompaa toteuttaa, alkoi tuottaa hyötyä nopeammin ja oli joustavampi alustan ja laitteen valinnan kannalta.

Avainsanat: mallipohjainen, testiautomaatio, kattavuus, vian havaitseminen

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

AAL	Adapter Action Language
ADB	Android Debug Bridge
API	Application Programming Interface
AUT	Application Under Test
fMBT	free Model-Based Testing tool
GUI	Graphical User Interface
ISTQB	International Software Testing Qualifications Board
MBGT	model-based graphical user interface testing
QA	Quality Assurance
QR code	Quick Response Code
SUT	System Under Test
TAE	test automation engineer
TUNI	Tampere Universities
URL	Uniform Resource Locator

1 INTRODUCTION

Robustness and reliability are important features for Android applications due to the competitive market. Test automation is a crucial part of quality assurance of modern mobile applications [7], and while the demand for test automation is high [48, p. 97], the use of test automation for Android applications seems scarce [40, 42].

Test automation conducted via the graphical user interface (GUI) of an application is one of the common ways to automate some of the recurrent but necessary testing tasks to relieve people from them. GUI testing is usually a part of regression testing, meaning that it is used to make sure new modifications to an application have not changed behavior in an undesired way. A traditional way of implementing GUI test automation is producing a script that runs the tests. Model-based testing (MBT) is a modern alternative to manual scripting that differs from it fundamentally in that it enables the option of automatic script generation based on a model.

In this thesis, the performance of model-based GUI testing is studied in terms of use case coverage and the amount and severity of faults found. This is done by testing an Android application while it is being developed using a model-based GUI test automation implemented with fMBT. As a point of reference, a manually scripted GUI test automation is also used on the same application over the same period. The idea is not to compare the solutions and find a "better" option but rather to assess the performance of MBT in the context of Android applications while pointing out the advantages and disadvantages of the technique. The possibilities of MBT in conducting extreme testing are also examined. Extreme testing, in this instance, refers to testing that requires extreme accuracy, stability, and continuity.

The goal of this study is to find out whether there is a significant advantage to be gained from applying model-based testing for an Android application, and if so, how and in what situations it may manifest. Also, the findings of this study may be useful for someone considering implementing GUI test automation for an Android application.

The results of this study indicate that the performance of MBT for an Android application is in line with the general theory on MBT. If enough effort is put to the implementation of MBT, great performance can be achieved with it in terms of both coverage and fault detection, but there are also downsides to using MBT. It is also found that MBT enables testing execution paths that are difficult for a human to test as well as executing extremely long test cases.

The theory of test metrics and test technique comparison, as well as the decision on the metrics used in this study, is presented in Chapter 2. The system under test (SUT) is described in Chapter 3. Chapters 4 and 5 delve into the two approaches of implementing test automation both in theory and by presenting the implementations of the test automation solutions created for this study. The results are analyzed in Chapter 6 and the conclusions presented in Chapter 7.

2 TEST METRICS AND TEST QUALITY ASSESSMENT

To be able to measure the performance of model-based GUI testing (MBGT) in comparison to traditional manually scripted GUI testing, a set of metrics has to be chosen. Some basic test management activities such as risk area identification, status tracking, and process improvement would be difficult without satisfactory metrics [17, Chapter 10].

The International Software Testing Qualification Board (ISTQB) defines a metric as "a measurement scale and the method used for measurement" [28] and measurement as "the process of assigning a number or category to an entity to describe an attribute of that entity" [29], the number or category being a measure. The book Systematic Software Testing by R. Craig and S. Jaskiel [17] derives term definitions from Dr. Bill Hetzel [26], the first being "measurement used to compare two or more products, processes, or projects" for metric, which is perhaps a more fitting description for the metrics used in this study. The second term, meta-measure, means "a measure of a measure". A meta-measure is usually used for measuring the effectiveness of another measure, for example, the number of faults found per inspection hour [17]. To be exact, the meta-measures are what is selected in this chapter but as mentioned and practised by Craig and Jaskiel along with many others [17, Chapter 10], it is not crucial to differentiate between metrics and measurement or to use the specified terms. Hence, in this thesis, the term metric may be used for measures, metrics, and meta-measures.

Some metrics commonly used for test automation evaluation or improvement purposes can be very precise such as the number of faults found or the amount of code covered by the tests. On the other hand, some of the factors that ought to be taken into account are also dependent on the evaluation of more abstract attributes such as the difficulty of learning to use a technique or the applicability of a technique to new situations.

2.1 Test quality assessment principles

In Jeff Tian's book Software quality engineering: testing, quality assurance, and quantifiable improvement [48, p. 285], three general questions related to the comparison of quality assurance (QA) techniques are presented: cost, benefit/effectiveness, and environment. Similarly, the framework proposed in the experimental study A Framework for

Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques by S. Eldh et al. [19] aims to compare test techniques basing on those three areas. The meaning of efficiency and effectiveness is very much in line with the meaning of cost in relation to benefit, as the idea in both cases is to assess what can be achieved by allocating a certain amount of resources into something. Generally, words that are used to convey pretty much the same meaning such as efficiency, effectiveness, cost-efficiency, cost and benefit in addition to practicality, environment and applicability are commonly present in papers regarding the comparison of testing techniques [3, 6, 8, 12, 19, 45, 48, 51].

The ISTQB Advanced Level Syllabus - test manager (2012) categorizes metrics into four categories: project metrics, product metrics, process metrics, and people metrics. Project metrics are used to measure progress toward a defined goal for the project. These may be, for example, percentages of passed and failed test cases. Product metrics, such as defect density, measure the attributes of the product that is being tested. Process metrics measure processes, for instance, the testing process, by measuring things such as the percentage of defects detected by testing. People metrics, on the other hand, measure the capability of people in different tasks, e.g., the number of test cases implemented by a tester or test group in a given time frame. It is noteworthy that according to the syllabus any one metric may belong in many of the four categories. [15, p. 38]

This section examines the theory of testing technique and, more broadly, QA technique assessment and comparison. The cost, effectiveness and environment division presented by Tian [48, p. 285] is used to separate the different principled basis for test comparison. As the purpose of this thesis is to study the performance of model-based testing paradigm, the focus is on the effectiveness related basis for comparison. The basis and reasoning for test metrics selection for this study are further explained in Section 2.2.

2.1.1 Environment-related basis for comparison

The environment-related question in comparing QA techniques mainly focuses on the applicability of a technique to a certain development or maintenance environment. Software testing, as one part of QA, is mostly used in development environments but as mentioned by Tian, it can also be applied to software maintenance. That is, in addition to evaluating and measuring the applicability of a testing technique to a given software development environment, its applicability to software maintenance can be assessed, for example, in locating bugs and verifying fixes to bugs reported by customers. [48, p. 285]

Tian [48, p. 290] approaches the applicability of a QA alternative from three perspectives: its applicability in a certain part of the development process, regarding the product under development and in terms of required participant expertise. On a more practical level, in [19] applicability is defined as "the ability for the technique to be automated and used in a

realistic and diverse context" and it is mentioned that "one aspect of applicability is generality which measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex code modifications, and real applications." In [20], the applicability issue is addressed as a question whether certain techniques are better suited for different environments. These environments could depend on programming concept and language, or certain types of software.

The applicability of a technique often seems to be closely tied with the cost of applying the technique. For example, [3] defines applicability as "how well and at what cost a technique/process/practice can be implemented in a given context to provide benefit". This definition twines together applicability and cost. Similarly, Tian [48, p. 290] considers required participant expertise as a direct factor on both the applicability of a technique and the cost of a technique.

Based on the references used in this section and a broader literature review, the environment-related question in selecting a testing technique is not an easily quantifiable problem. On the contrary, determining the applicability of a technique seems to require a lot of estimation, especially when the applicability of a technique to a novel use case has to be assessed. In addition, it appears that one has to be careful when using both applicability and cost as the basis for test technique selection because common factors seem to directly affect both questions. Mistakenly taking some of these factors into consideration twice could result in a heavily unbalanced assessment of a technique in comparison to the intended one.

2.1.2 Cost-related basis for comparison

The cost of a testing solution most usually refers to the time and effort required from software professionals to implement and perform the testing, as well as the cost of the equipment that is required to perform the testing [2, 3, 48, p. 296]. Training project participants, the time reserved for meetings, other overhead, and the acquisition and support for required software tools are issues that Tian mentions as possible indirect costs. Tian also presents two key factors affecting the above-mentioned costs: the "simplicity of the techniques associated with the specific QA alternative", which makes learning and training the use of such techniques easier, and the "availability of tool support", which usually saves time and effort of the people applying the testing. In short, cost is usually measured by the required monetary and timely resources. This seems to be a common approach [51, 48, 50, 2], some even taking it to a level where the return on investment of implementing and using a technique is calculated [3].

According to [48, p. 296], the cost of a QA activity for a traditional QA technique like testing which focuses on the detection and removal of defects can be directly related to detecting and fixing faults. Besides, as mentioned by Basili and Selby in [12], when the effort required for a technique to help detect and fix bugs is examined, the isolation costs

are to be taken into account in addition to detection costs. Although the instance in [12] is an individual case in that it compares code reading to functional and structural testing, the same principle can be applied to testing in general: the effort required to locate a fault after its detection should be considered as part of the cost of fixing the fault. Therefore, the better a technique is at isolating faults, the smaller should be the cost of locating the faults.

An empirical study [8] that compared model-based and dynamic event-extraction based GUI testing techniques focused partly on the cost of test generation and execution of the techniques. The cost was measured by two variables, cost in terms of the numbers of executed events and cost in terms of elapsed time. In this instance, manual human effort in setting up the testing environments was not considered a key factor when assessing the cost of a technique. The argument for omitting that effort was that the time required for doing so was small when comparing the trends in the costs of the techniques. However, the costs of configuring an application under test (AUT) and testing tools are taken into account in the study [8]. Consequently, it is mentioned that the amortization of these costs by applying automated test approaches repetitively should be considered by testers using automated GUI tools.

It is underlined in [48, p. 296], that the time at which a defect is found is essential when assessing the cost caused by the defect. This is justified [48, p. 296] by the fact that in addition to having to fix the original fault later on, there is the possibility of other faults having been caused by the original one and the cost of fixing these is dependent on when the original fault is found. Therefore, the time at which a defect is found could very well be considered a factor of cost.

To summarize, a common way to use cost as an evaluation principle for a testing technique seems to be the measurement of resources in terms of money and time required to implement and perform the testing. On a broader level, there are other indirect costs that can be regarded as dependent on the used testing technique. These include, for example, the cost of detecting a defect later rather than sooner, and the cost of fixing found defects based on the information provided by the used technique. Another example of something often perceived as a cost is the amount of time that it takes for a test automation run to finish.

2.1.3 Effectiveness-related basis for comparison

Tian points out in his book [48, p. 286] that since different QA alternatives exist each for their own reason their benefit comparison should be made on a situational level rather than as an assessment of the general benefit they provide. He approaches the benefit analysis of a QA technique from four viewpoints: defect perspectives, problem types, defect levels, and pervasiveness plus constructive information and guidance for quality improvement [48, pp. 291-295].

Defect perspectives can be separated into two parts: the observation of a defect during a QA activity and the types of follow-up actions that are taken after the observation. For testing, the observation consists of observed failures and the follow-up action is fault removal. In other words, detecting failures and making, as characterized by Tian, "analyses based on information recorded during the failed executions to locate and remove the underlying faults that cause the failures". [48, pp. 291-292]

Problem types viewpoint on the effectiveness comparison of QA alternatives refers to the effectiveness of a technique in dealing with different kinds of problems, from errors or error sources to failures of different severity and simple faults. Defect prevention techniques, for example, are effective in finding systematic errors or conceptual mistakes whereas testing usually proves especially useful in finding dynamic failures and related faults. Fault tolerance techniques, on the other hand, are effective in dealing with small operational failures. The explanation to the difference in the types of defects usually detected by different techniques is based on their nature: a human inspector of code can easily focus on a very small sample of code and detect localized faults, but it is a lot harder for a human than for a computer to monitor the complicated interaction between multiple components over time. [48, pp. 292-293]

Defect levels and pervasiveness is an area that should be taken into account in QA technique comparison. For example, defect prevention is the most effective QA technique in finding systematic or pervasive problems that could be causing high defect levels in an organization [48, p. 293]. This makes sense because defect prevention aims to reduce the number of defects produced whereas testing, for example, is more of a reactive technique. On the other hand, incidental problems are usually better dealt with by using other techniques than inspection which, according to ISTQB glossary, is a formal review attempting to find issues in a product to improve development processes [30] or defect prevention [48, p. 294].

By constructive information and guidance for quality improvement, Tian refers to the ease of result interpretation and the production of useful general information by a QA technique. The interpretation of results is easier for inspection than it is for testing and significantly harder for other QA alternatives. The ease of result interpretation can be assessed by the amount of effort required before being able to take follow-up actions. [48, p. 295]

While Tian [48] gives us a general overview on the issues that should be taken into account when comparing the effectiveness of QA alternatives, several papers [15, 12, 50, 52, 8, 6] have used more detailed metrics that have been used to determine the effectiveness of a testing method or the quality of a product. Two categories of these metrics that are related to one another and that are relevant in terms of the goals set for test automation in this study are further examined: defect/fault-based metrics and coverage-based metrics.

Defect metrics

Finding defects is one of the main reasons for using the kind of test automation that is studied in this thesis. Therefore, it is reasonable to have a focus on the defects found by a solution when assessing its performance. Examples of measurement criteria include the number of defects/faults found/failures caused [50, 12, 45, 8, 52, 3], the type/severity of found defects [50, 12, 52], the time spent until defect/fault is found [12] or the percentage of faults detected [12, pp. 1286-1287, 20].

An experimental study, Studying the Characteristics of a "Good" GUI Test Suite by Q. Xie and A. Memon [50] evaluates the fault-detection effectiveness of test suites with different characteristics. Another experimental study comparing three different testing techniques, Comparing the Effectiveness of Software Testing Strategies by Basili and Selby [12], uses three fault-related categories of measurements: fault detection effectiveness, fault detection cost and the characterization of faults detected. These are further divided into subcategories such as the number of faults detected and the percentage of faults detected. The study by Basili and Selby also mentions coverage in terms of program statements tested as a possible factor in the ability of a technique to detect faults and the programming expertise of the testers as a possible factor favoring certain types of testing techniques [12, p. 1287]. Engström et al. [20, p. 24] describe a safe regression test selection technique as a technique with which all defects found with a "full test suite" are found.

Coverage metrics

Coverage-based metrics are used to measure the extent to which the SUT is tested or how much of the SUT is covered by the tests. Coverage can be measured in multiple dimensions varying from code coverage to, for example, usage scenario coverage. Code coverage typically measures the amount of code that is tested by the tests whereas usage scenario coverage measures the amount of or perhaps the percentage of total possible scenarios that are tested. Examples of coverage-related test assessment criteria used or presented in literature include, for example, code coverage [27, 39], which could be based on lines or blocks of code executed in automation, for example, requirements coverage [13, Chapter 8.1, 38, p. 389], which means coverage of a selection of requirements and statement/path/transition coverage [12, 8, 6, 13, Chapter 8.1], which usually refers to coverage in terms of different ways to execute code or coverage in terms of reaching possible states. These coverage terms seem to be used in various ways depending on the paper they are presented in and sometimes they overlap or seem to be used as synonyms. For example, a study by R. Lincke et al. [43] comparing software metrics tools found over 200 metrics with different names which together actually described only 47 different metrics.

The article GUI Interaction Testing: Incorporating Event Context by X. Yuan et al. [52]

describes the different sequences of events, e.g., a user tapping a cancel button causing a software to end up in a certain state which may affect how the software executes, as a problem for GUI testing as some techniques only test a small subset of those states. The article also highlights that the order of events and the insertion of an extra event in a combination of events may make the difference between the events triggering a fault or not. To address these considerations, the article presents new coverage criteria that consider the possible start and end positions for every event, the sequence length and the event combination strength [52].

2.2 Metrics Selection

Firstly, it should be noted that a lot of the types of metrics addressed in Section 2.1 may not be applicable or well suited for comparing different test automation techniques. The term test metrics very often seems to refer to metrics that are used to measure the quality of a product, as an example, the failure rate of test cases caused by defects in the product. Another common use case of metrics is the measurement of the progress of a testing process, for example, the number of test cases. The failure rate of test cases or the number of test cases are not suitable for the comparison of two techniques that differ on a conceptual level and do not share the same meaning of terms like test case. It is also noteworthy that the theory by Tian [48] is mostly asserting general guidelines for the assessment and comparison of QA activities while test automation is only a small part of one of those activities. Still, there are plenty of metrics and applicable principles mentioned in Section 2.1 that can be directly or indirectly used to assess the qualities of different testing techniques so that their performance in certain areas can also be compared.

It appears to be widely accepted that metrics goals should be the leading factor in metrics selection [17, 18, 41]. In other words, the popularity or the general effectiveness of a test metric should not necessarily be prioritized over the suitability and fitness of another, perhaps a less popular one, for the desired measurement purposes. The primary goals for the UI test automation studied in this thesis is to provide help in finding faults in the SUT, confidence in the reliability of the SUT, and assistance to manual UI testing in verifying the correct functionality of new features and detecting defects. The secondary goals are to reach the primary goals as cost-efficiently as possible and as vastly as possible, vastness meaning the ability to perform test automation on multiple platforms and devices. An additional desirable high-priority goal is the capability to perform the type of testing that is extremely difficult or laborious to perform manually, for example, testing to find memory leaks, errors caused by actions taken with specific time intervals and slight misplacement of UI elements that may later prove critical.

One of the Oxford Advanced American Dictionary definitions for the noun performance is "how well or badly something works" [46]. Thus, as the goal of this thesis is to study how well or badly model-based testing works for an Android application, the used metrics

(or meta-measures) should reflect how well or badly the technique reaches the goals set for test automation in the previous paragraph. The metrics are divided to primary metrics and secondary metrics based on the similar division of the goals for test automation.

Primary metrics

To measure how well model-based testing detects faults in an android application, the number of faults detected by the technique was recorded and the type or severity was analysed. The results of the manually scripted test automation were used as a reference. Therefore, the detailed defect-related metrics could be described as the number and the severity of the faults found by model-based test automation over 3 months in relation to those of the faults found by manually scripted automation over the same period of time. The severity was estimated on a scale of 0-9, 0 being the highest.

One of the main goals for the studied type of test automation was to provide constant oversight over the system under test, reassuring the development team that the product under development is generally speaking working, recent modifications have not caused regression and to notify the development team fast if they have. To better accomplish this goal, the test coverage should be high. It is mentioned in [15, p. 39] and [48, p. 94], that coverage can be used as a surrogate metric for confidence. Another reason to measure coverage metrics is that defect detection and coverage effectiveness do not necessarily correlate, as pointed out by Bae et Al. in [8, p. 45] and by Basili and Selby in [12, p. 1287]. Therefore, the third selected primary metric was the use-case-based coverage that could be reached with the technique, and more precisely, that could be reached with the technique in the time that was used and the level expertise available. A more exact but less intuitive name for the coverage criteria would be coverage in terms of *test steps* which are presented in Section 5.2. As with the defect-related metrics, the same type of coverage analysis for manually scripted test automation was used as a reference.

Secondary metrics

Because the secondary metrics are somewhat interconnected with the primary metrics, it was necessary to try to mitigate the effects of uneven effort put to the solutions. Therefore, one of the goals of this study was that the amount of effort put to the development of the used testing methods is roughly the same. In that sense, the number of faults found and the coverage provided by each of the test automation solutions was achieved with roughly the same cost. However, the cost was not accurately observed and accredited for each of the solutions. This is better examined in Chapter 6.

The significance of cost is also diminished by the fact that the expertise of the test engineer implementing the solutions was the same for both. However, the perceived use-

fulness of model-based testing and the manually scripted test automation could be assessed at different points of implementation, therefore giving an idea on the performance in relation to the effort that had been put in at different points. In addition, assuming that mitigating the cost factor worked, the performance in terms of other metrics should be a fairly good indicator for cost-effectiveness.

Spending roughly the same amount of effort on both implemented testing solutions also means that the impacts of the techniques' quickness and easiness should be taken into account when assessing what could be achieved with a certain cost. Therefore, the applicability of model-based test automation in comparison to the manually scripted automation was estimated. This was done based on perceived difficulty and required expertise, platform- and device compatibility, and maintainability.

3 SYSTEM UNDER TEST

The system under test is a mobile application named PiceaOne, developed by Piceasoft Ltd. The application is a solution for mobile device verification and diagnostics. Only some of the features of the part of the application that is used for diagnosing mobile devices are tested in this study. The application requires the use of several APIs to function correctly and therefore also the correct functionality of these APIs is indirectly tested. It is important to note that in this chapter the words referring to test entities such as test case, test set, and test operation refer to the tests performed by PiceaOne on the device diagnosing. Referrals to the UI test automation tests that were implemented for this study are explicitly mentioned.

This chapter explains PiceaOne application on a conceptual level and gives an idea of how the application is used. This is necessary as it is essential in differentiating the paradigms of model-based testing and manually scripted test automation later in the paper. The screenshots used in the figures were all taken with Samsung Galaxy S8, the device that is used in this study. The red rectangles on them are not part of the application, they have been edited afterward to pinpoint areas. This chapter also clarifies the constraints that were made to the tested features and explains why these constraints were made.

3.1 PiceaOne Diagnostics

The idea of PiceaOne Diagnostics is that the user can use the application to run tests on the device software and hardware. The tests are either automatic or interactive and they can be run as singular units or in larger collections. Some of the test cases need to be run one at a time and others can be run in parallel. By using the application the user can map out which parts and features of a device are defective and what is the general condition of the device. The application also presents device information and informs the user of found issues and possible solutions to the issues.

A test run can be started by opening the application and reading a QR code that envelops the information defining which tests are to be run or the application can be directed to run certain tests in real-time using a web interface or a PC application developed by Piceasoft Ltd. A third way, the one used in the test automation purposes in this study, is to start the test run by opening the app using Android Debug Bridge (ADB) and giving it the

information on what tests to perform in a deep link.

3.1.1 Operations, Sets and Cases

A test run, also referred to as test operation, in PiceaOne consists of one or more test sets which in turn consist of one or more test cases. When an operation is started, the execution of test sets starts in a predefined order, as well as the execution of the test cases included in the test sets. In other words, the execution order of every test case and test set is always known before the operation. Exceptions to this are the test cases that are run in parallel as it is possible that, for example, GPS location data is obtained faster than mobile data connection while the mobile data test case is higher in the execution order. However, this variability only affects a small number of test cases, none of the test sets and it is not expected to affect the results or events in the application apart from the order in which they occur. The predefined test execution order is also independent of the test selection, the test selection order and how the test operation is started. Therefore, even if some test sets or cases are omitted or tests are rerun, the expected order of execution remains the same with the omitted sets or cases not executed.

Each of the test cases has a result after it has been run. The possible result values in descending order of priority are *canceled*, *failed*, *skipped*, *passed with issues*, and *passed*. Each of the test sets also has a result, the value of which is dependent on the result values of the test cases of that test set. The test set result is determined by the highest priority result value that has realized for the test cases of that set so that if the results for audio test set cases are for instance *failed*, *failed*, *passed*, and *canceled*, then the result value for the audio test set will be *canceled*. If the results for the audio test set cases are *passed with issues*, *passed*, *passed*, and *passed*, the result of audio test set will be *passed with issues*. Only test cases that have been started will have results meaning that if a test set is canceled before some cases are started, those cases will not have results. However, in case some test cases have been run and are then part of a retried run, even if the rerun is canceled before those cases, they will still have their results from the previous run.

3.1.2 Views

The views in PiceaOne that are not directly related to test execution include the opening/-main view, the version information view and the QR code reading view. The opening view is shown when the application is opened and from there the user can choose to open the QR code reader or the version information view. Returning to the opening view is the only option in version information view and in the QR code reader view the user can also read a QR which starts a test operation.

While the Diagnostics operation is ongoing the view(s) of the ongoing test cases are

shown as they are being executed. If test cases are run simultaneously, one view may be shown for several cases or one test case may be executed in the background while the view for another case is shown. In each of these views, there is the possibility to open an advisory view and return to the ongoing case(s) view. Between test case views and test sets views, an operation summary view is shown of the list of test sets so far executed. The operation summary view is also shown during general and software test set execution and after an operation is finished.

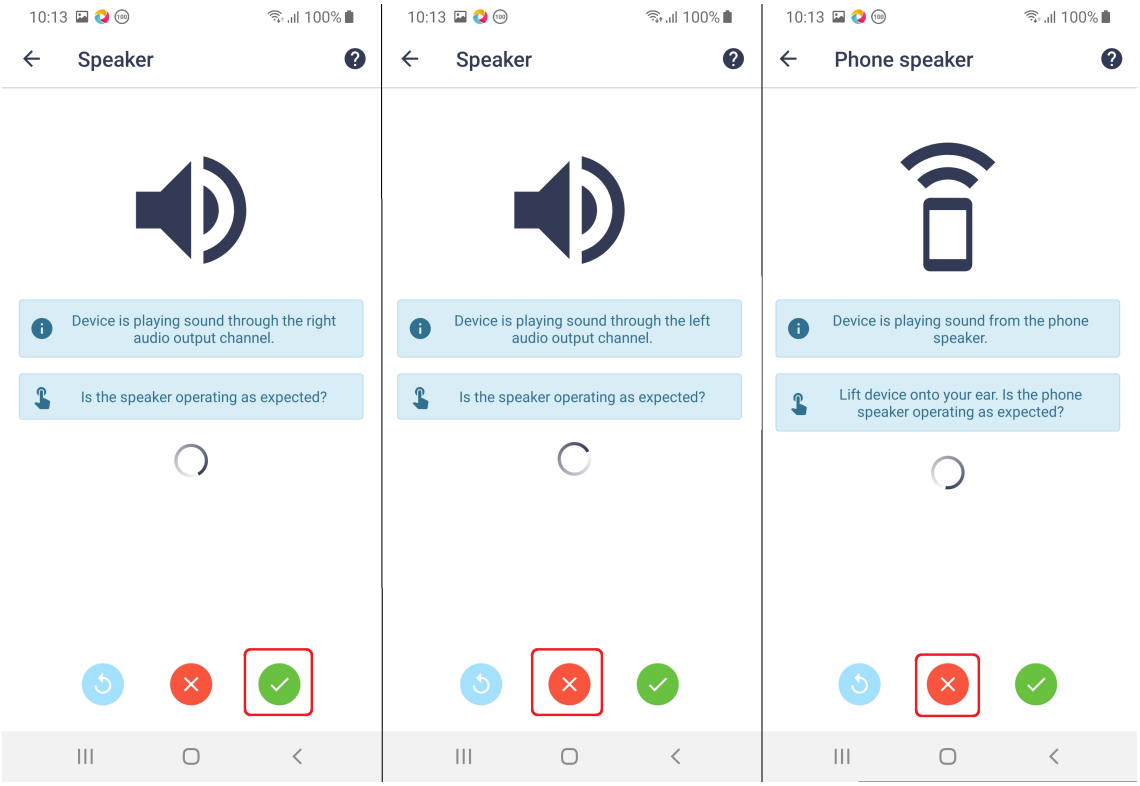

Figure 3.1. *Three screenshots presenting some of the views that are displayed during the execution of the speaker and phone speaker test cases on PiceaOne.*

Figure 3.1 presents the execution flow of the first two test cases of the audio test set. The execution order is from left to right and the taps performed to proceed to the view in the next screenshot are indicated by the red rectangles. From the view of the third screenshot in Figure 3.1. the execution continues to the first screenshot of Figure 3.2. In the flow in Figure 3.1 the user passes the first part of the speaker test by tapping the green pass button and fails the second by tapping the red X button. As mentioned on the views, during the first part of the speaker test voice is played through the right audio output channel of the device. In the second part, the sound is played through the left channel and in the phone speaker test case through the phone speaker. In a real-world use case, the user is expected to listen and make a decision on whether the parts work or not based on the sound quality or absence of sound being played during each of the views. In the UI test automation, the aim is to perform UI testing on each of the PiceaOne test cases as comprehensively as possible, which means passing, failing and canceling PiceaOne test cases without paying attention to whether the speaker happens to work on

the device or not. This would not make a lot of sense anyway because the working and defective parts of the device are already known and the scenarios in which different parts are not working are to come in the real use of the application either way. At the rightmost screenshot of Figure 3.1, the Phone Speaker test case is failed.

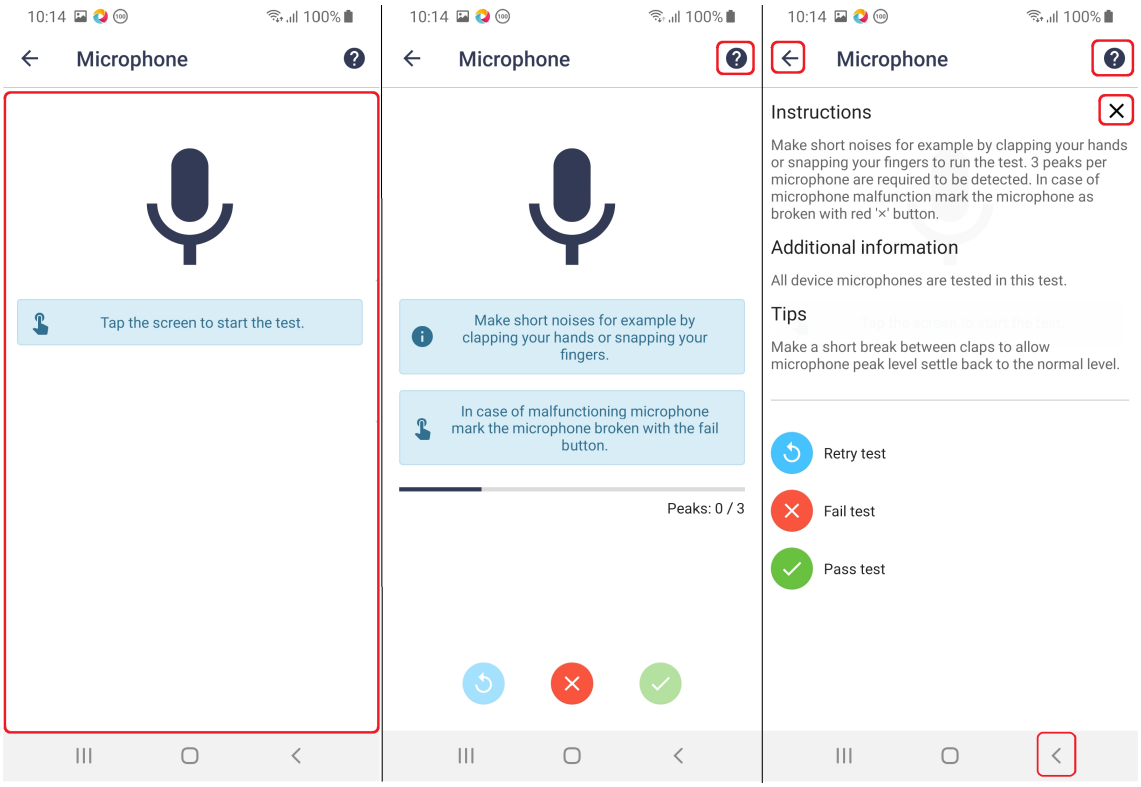

Figure 3.2. *Three screenshots presenting some of the views that are displayed during the execution of the microphone test case on PiceaOne.*

In Figure 3.2, the execution flow of the audio test set continues with the microphone test case ongoing. In the first screenshot, the UI area indicated with the red rectangle is tapped to start the test. After that, there is a view informing the user that microphone is being calibrated but that view is temporary, does not require user action and has been left out of the figure for practical reasons. After the device microphone has been calibrated to adjust to the surrounding soundscape the view in the middle of Figure 3.2 ensues. At this point, the application waits for noise peaks to be registered by the microphone. There are three ways to proceed: passing the test case by making three noise peaks near it and failing or canceling the test case at any point by either tapping the red fail button or the left arrow at the top left corner of the view. Although the speaker and phone speaker test cases are also interactive test cases, passing the microphone test case would require significantly more effort than they do. The problems related to this effort and why it was avoided are specified in Subsection 3.1.3.

In the middle screenshot of Figure 3.2, the user taps the help page button which opens the informative view for the microphone test. There is a similar informative view for every PiceaOne test case that has a UI. These informative views are meant to provide additional

guidance and tips. The closing options in the informative view are indicated by the red rectangles in the rightmost screenshot. Tapping them returns the user to the view of the test case that is being executed, back to the middle screenshot in this case. However, the execution of test cases also continues in the background while the informative view is shown and it may be closed automatically if, for example, three noise peaks are registered while it is showing for the microphone test.

After a test operation is finished the user can navigate between a device information view and the operation summary view. By selecting a test set in the test operation summary view, a similar test set summary view is shown presenting the executed test cases of that set and their results. By selecting a test case in the set summary view the user can open an issue-resolution view which shows the possible issues and solutions to them. An informative view about issues and resolutions can be opened from the issue-resolution view. Returning from each of the last mentioned four views opens the previous view.

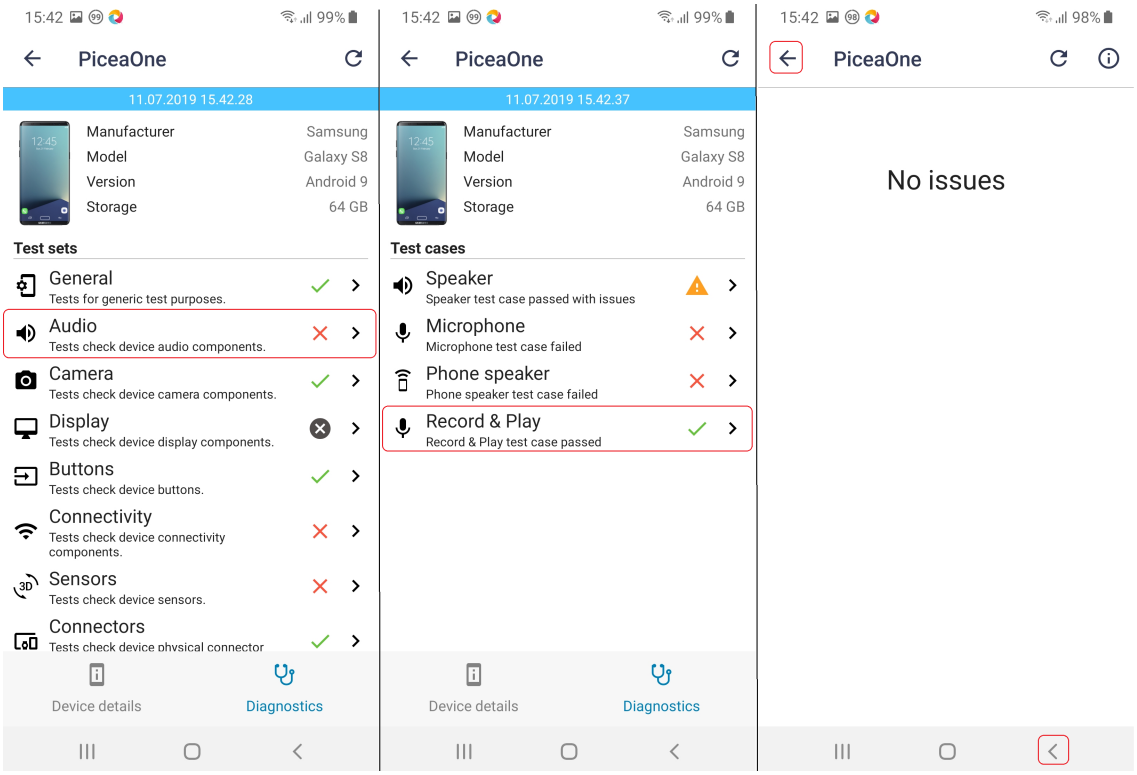

Figure 3.3. *Three screenshots showing how the result of an individual test case can be checked.*

Assuming that after performing the test cases like in Figures 3.1 and 3.2, a user of PiceaOne goes on to fail the microphone test case and pass the record and play test case, thereby finishing the audio test set. Assuming also that the audio test set was run as part of a bigger operation, the leftmost screenshot of Figure 3.3 is the view the user is shown. The view contains the results of the test sets depending on what were the results of their test cases. The picture and information of the device is shown at the top part of the screen, the button to open device details is at the bottom left corner, returning

to home can be done by tapping the left arrow at the top left corner and the operation can be retried by tapping the button at the top right corner. The executed test sets and their results are listed under the device information. The audio test set result is failed according to the logic specified in the second paragraph of Chapter 3.1.1. By tapping the area highlighted in Figure 3.3, the test cases and results for the audio set are shown as in the middle screenshot and by tapping the area highlighted in that screenshot the issue-resolution view for record and play test case is shown. There are no issues mentioned which makes sense because the test case was passed. Tapping the buttons highlighted in the view of the rightmost screenshot the user is taken back to the middle screenshot view.

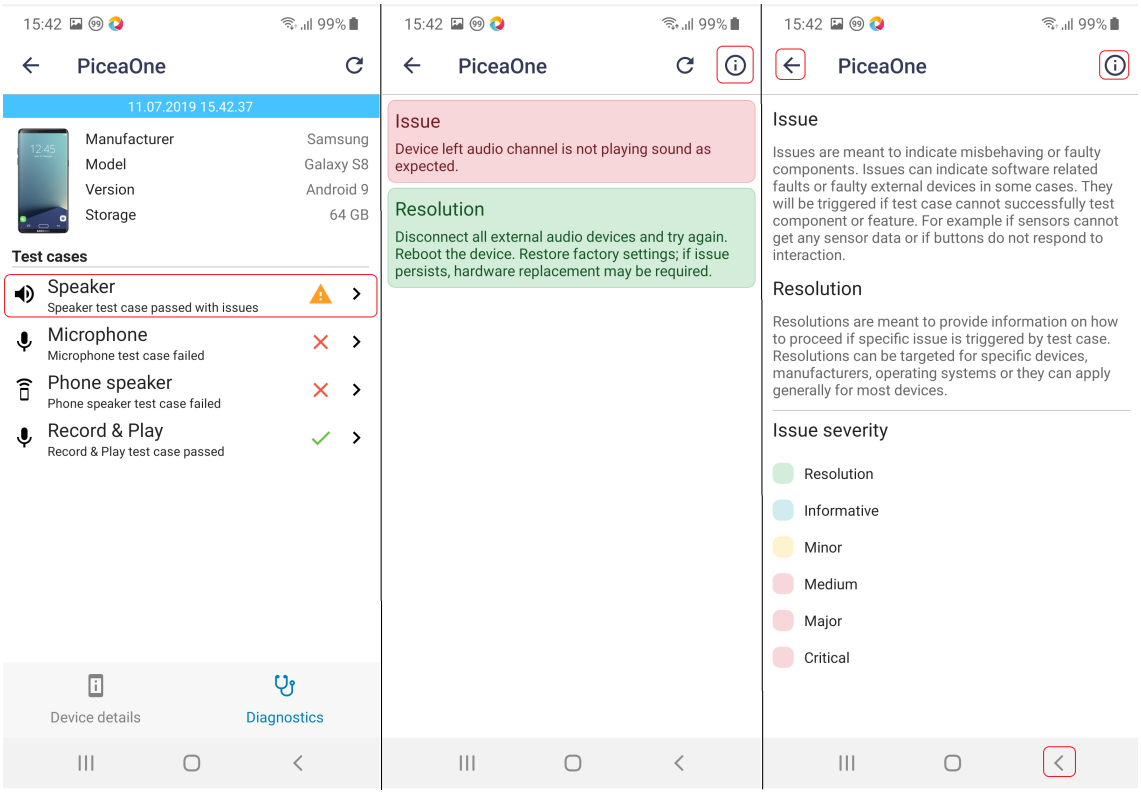

Figure 3.4. *Three screenshots showing how issues and resolutions are displayed and their meaning explained to a user in PiceaOne.*

The leftmost screenshot in Figure 3.4 is from the same situation as the middle one in Figure 3.3. However, this time the result is checked for speaker test which was passed with issues as the results were given so as if only one of the two audio channels was playing sound. The middle screenshot in Figure 3.4 shows an example of a PiceaOne issue and resolution pair that is fetched from a database using an API. The third screenshot is an informative view of issues and resolutions. Once again the highlighted areas indicate where to tap to proceed to the next (previous in the third one) screenshot.

The rest of PiceaOne Diagnostics works more or less like the audio test set does: some tests are either executed automatically or require user action, and their results are presented after the run. There are also subtleties in different tests that have to be taken into

account when implementing UI test automation.

3.1.3 Constraints subjected to GUI testing

PiceaOne is a fairly complex application with a lot of features. Therefore it was necessary to make constraints on the features and actions of PiceaOne that were tested in this study. This was done due to limited resources in terms of both time and labor. The testing of many of the features of PiceaOne would have required significant monetary resources in terms of equipment and programming. For example, running the sensors tests or connectivity tests both passing and failing all of them is extremely difficult to perform automatically because to do this the device settings have to be changed, SIM-card inserted and removed, device movement arranged, etc. As the main goal of this study is to study the performance of model-based testing as a paradigm, it was considered sufficient to test as many parts of the application as feasible without having to spend disproportionately much effort on the execution of individual test cases every way possible. This was done so that the tested areas were pretty much the same for both test automation approaches.

Feature constraints

Figure 3.5 is a screenshot from a website designed for building QR codes and links with definitions of test operations. The PiceaOne test sets that are tested in this study are selected in the picture and they each contain multiple test cases. The audio, connectors and battery test sets are expanded in the figure, showing the test cases included in those sets.

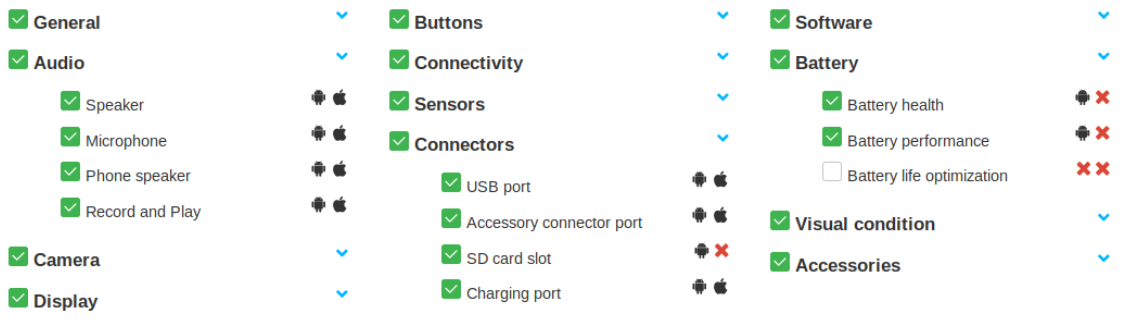

Figure 3.5. *The PiceaOne test sets that are tested with the cases for audio, connectors and battery test sets also shown.*

All in all, there are 14 test sets and over 73 (05.08.2019) Android-supported test cases available in PiceaOne. The automated UI testing for some of them was left out because the implementation of two test automation solutions was already time-consuming. Also, reruns of individual test sets and cases after being done with an operation, as well as the running of operations consisting of a single test case was left out of the scope. This was

done mainly because of the high cost in scripting and modeling complexity in comparison to perceived benefits. In short, the testing was focused on the most common ways of using the application. Still, 62 PiceaOne test cases in total are tested with the UI test automation solutions implemented in this study.

Equipment constraints

The main constraint on the used equipment was that only one device was used for testing. Even though the manually scripted test automation was running tests on multiple devices at the time of this study, the implementation of the model-based solution for multiple devices and platforms was considered inefficient in this study, partly because the focus is on model-based testing as a paradigm. Therefore only the tests performed on the same device were taken into account in this study. No one device supports all of the test cases for Android, thus some were left out because of equipment constraints. The differences in the testing approaches' applicability are better examined in Chapter 6.4.

The automated testing of features such as the camera autofocus test case or ambient light sensor test case could have been performed more thoroughly by using equipment to move the device from a position pointing to a QR code to a position not pointing to one, or from a bright environment to a shadier one. Also, some similar actions triggering the passing of some test cases could be programmed. However, the cost of this kind of comprehensive test automation was considered disproportionate to the benefit of being able to pass some test cases in addition to failing and canceling them. Furthermore, the scope of this study was also a limit. In short, the purpose of test automation, especially in this case, was to provide assistance to manual UI testing cost-efficiently, not to attempt performing every possible action automatically.

3.2 Used hardware and platform

The UI testing that is taken into in this study was performed on a physical Samsung Galaxy S8, 64 GB device running Android Pie. The tested application was a beta-version of PiceaOne that was updated daily before each UI test run. The manually scripted test automation was run on a Mac mini running macOS Mojave 10.14.6 and the model-based solution on an HP Elitebook running Ubuntu 19.04. USB-cables were used for the communication between the device and the computers. To enable the computer control of the device, USB-debugging mode had to be turned on and accepted in the device developer options.

4 TRADITIONAL SOFTWARE GUI TEST AUTOMATION

The ISTQB glossary [31] defines test automation as "the use of software to perform or support test activities, e.g., test management, test design, test execution and results checking". Axelrod [7] mentions that it is difficult to exactly define all the possible meanings of test automation and gives "using software to help in testing of another software" as his best shot of a generic definition but also gives a more detailed definition in Chapter 6 of his book stating that an automated test can be defined as a computer program that gives inputs to another and monitors the feedback, comparing it to expected feedback and presenting the results of the comparison. Either way, software GUI test automation can be described as test automation that uses interactions with the GUI of the SUT. The usual rationale for applying GUI test automation is that it reduces manual testing effort and therefore the time and cost spent during each stage of development [10, p. 72, 7, 14, pp. 8-9].

A common way of applying GUI test automation is designing, implementing and executing test scripts that perform some predefined sequence of events on the UI while also monitoring the UI and verifying that correct elements are shown. This chapter briefly delves into the theory of performing scripted test automation and introduces the implementation of manually scripted test automation used in this study. Manually scripted test automation in this instance refers to test automation in which the test automation engineer (TAE) manually takes part in the creation of the test scripts, albeit using a set of tools as assistance.

4.1 Scripted test automation theory

The ISTQB Certified Tester Advanced Level Syllabus - test automation engineer [9] states that test cases need to be defined as sequences of actions that are performed on the SUT. In addition to the actions, the test data that is used in interacting with the SUT and the verification steps that verify the correct functionality of the SUT need to be defined for test cases. The sequence of actions may be derived from a procedure or be implemented in a test script. [9, p. 31]

The syllabus [9] presents four ways of producing the sequence of actions with varying

levels of abstraction and automation: implementing test cases directly into automated scripts, designing procedures that can be transformed to automated test scripts, using a tool to transform the procedures to test scripts and using a tool to generate automated procedures and/or translating test scripts directly from models. The implementation of test cases directly into automated scripts bears a heavy load in maintenance and lacks automation, and is therefore, the option that is the least recommended by the syllabus. The design of procedures and transforming those to test scripts adds a little bit abstraction to the direct implementation but also lacks the automation in the generation of scripts. The third alternative, in which the TAE directly takes part in producing test scripts by using a tool to turn procedures into test scripts, combines abstraction and automation. [9, p. 31]

The SUT project characteristics and the difficulty of implementing the action producing options is noteworthy when selecting one because while the options that have a higher level of abstraction and automation may be superior in principle, the simplicity and fast implementation possibility of the less advanced options are advantages in terms of required expertise and short term value [9, p. 31]. Model-based generation of test scripts is further examined in Chapter 5.

4.1.1 Matters to consider when implementing test automation

There are several areas of assessment listed by Arnon Axelrod in [7] that should be taken into consideration when implementing a test automation solution. Some issues come to mind fairly intuitively but others one may not think about at first thought. These are issues such as the preciseness, maintainability, their combination sensitivity to change, failure handling, test case length, test case dependencies, logging and evidence collection, and trust [7].

Logging and evidence collection refers to saving logs and evidence, for instance screenshots, of the test run and possible failures. This is done to ease the determination on whether in failure cases just the test code was defective or there was an actual fault. Trust refers to the mistrust between developers and testers, as well as between humans and test machines. To build trust, the testing should mostly fail due to real bugs instead of test code errors and while it is understandable that it misses some bugs, it should not miss bugs that it was designed to catch. Failure handling is related to how the testing handles unexpected situations, bearing in mind that not only bugs but also intentional changes, environmental issues such as network issues and occasional incidents, for example, scheduled notifications or accidental human contact, may cause test failures. To prevent little issues from preventing the execution of large parts of automated testing, Axelrod encourages defining short and focused test cases that are not dependent on the successful execution of one another. [7]

Based on the frequency of references to it, the preciseness of testing, the maintainability of the tests and their combined effect, the test automation's sensitivity to change is one

of the most essential factors in test automation development [7, 9, 25] syllabus pages 31-36. The preciseness of test automation is essentially obligatory as computers can not deal with vagueness. The maintainability of automated tests is related to the fact that the same code does not have to be tested twice so the tests are expected to be run on a changed SUT, which means that the tests often have to be updated to match the new version SUT. A test automation solution's sensitivity to change depends greatly on how the issues of precision and maintainability are handled. The modularity of the solution provides a lot of flexibility to the balance of preciseness and maintainability. [7]

4.1.2 Scripting approaches

A. M. Jonassen's book Guide to advanced software testing [25] approaches test automation setup from a tools perspective. Multiple test script related tools from test design tools to test execution and keyword-driven automation tools are mentioned in the book [25, Chapter 9.3.3]. A similar perspective is taken in the book Verification, Validation and Testing in Software Engineering [10, p. 85] by Bandeira et al., although in this case GUI testing tools are wrapped under capture/playback tools with the addition that the scripts produced by these tools are best used when modified. There are also several well-established approaches for test case automation listed in the ISTQB Syllabus for TAEs [9, pp. 31-36]. These approaches, with the exception of model-based testing which is discussed in Chapter 5, are next clarified based on the two sources in ascending order in abstraction and automation.

Capture and playback approach

When test automation is created using the capture/playback approach, usually a sequence of interactions manually performed by a user of the SUT is recorded. Another way to do this could be for instance recording network traffic such as HTTP requests [7, Chapter 2]. The outputs of the SUT may also be recorded for later use in verification steps. The actual testing can be performed with varying levels of automation and detail, varying from a tester watching the SUT and verifying the correctness of the outputs to more or less detailed automatic monitoring of the outputs either over the whole playback or in specified checkpoints. Jonassen also describes capture and playback as the basic principle for test execution tools in general in the sense that a script is defined and then the recorded script is executed, as the reactions of the SUT are monitored for anomalies [25, Chapter 9.3.3.5] and goes on to describe editable scripts basically as advanced versions of this. The capture and replay approach can be used when the TAE directly implements test cases into scripts. [9, p. 31]

According to the ISTQB syllabus, the main advantage of the capture/playback approach is the ease of implementation and use. On the other hand, capture/playback is vulnerable to

changes in the UI of the SUT, for example, and may, therefore, cause high maintenance costs. The vulnerability to changes and the possibility of having failures in recorded scenarios is a weakness emphasized also by Axelrod [7]. Additionally, capture/playback can only be implemented after the SUT has been made available. [9, pp. 31-32]

Linear scripting and structured scripting

Linear scripting is very similar to record and playback in that a test tool is used to record interactions with the SUT and if necessary, the outputs too. An essential difference is that when linear scripting is used, the recorded test case scripts can be edited and comments added to enhance readability. Generally, a larger script is generated by recording for each procedure and the scripts are then edited by adding more checks, for example. These scripts can then be rerun by the tool. When advancing from linear scripting to structured scripting, according to the ISTQB syllabus [9, pp. 32-33], the difference is that structured scripts make use of script libraries that often include useful, reusable scripts.

As with capture and playback, high maintenance and scaling costs, especially when having to create numerous test cases and when testing software with numerous releases is the downside of linear scripting [9, pp. 32-33]. This eases up when advancing to structured testing, however, at this point programming skills are required because simple recordings are not enough to employ the advantages of structured scripting [9, p. 33]. According to Jonassen, the newer versions of test execution tools like these that support script editing and fully coded scripts are replacing capture and playback tools indeed because of the higher maintainability [25].

Data-driven testing

Data-driven testing is a step forward from structured scripting. The essential improvement in comparison to structured scripting is that the inputs are separated from the scripts to data files. It follows from this that a test script, typically called the "control script", can be multiplied in different versions that use the data from the data file(s) to create a high number of test cases. [9, p. 34]

Data-driven testing is not described in quite the same detail by Jonassen. However, her Guide to Advanced Software testing [25] mentions keyword-driven automation tools' ability to use parameter-driven test scripts in Subsection 9.3.3.6, which could be interpreted as something similar. On top of that, separate test data preparation tools are presented in Subsection 9.3.3.2. These tools are said to include the selection of data, for instance from a database, data creation, generation, manipulation, and editing. [25]

The two main advantages of data-driven testing are cost reduction in the adding of new tests and the ability to deepen the testing by automating multiple variations of useful test cases [9, p. 34]. The possible pitfalls for data-driven testing are the necessity of manag-

ing readable data files and the possibility of accidentally leaving out critical 'negative test procedures' when too much focus is on data and hardly any on the procedures [9, p. 34]. The ISTQB glossary defines a test procedure as "a sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution" [32]. The main advantage of using data preparation tools is the capability to large amounts of data and the downside is the possibility of the tools creating loads of unusable data. [25, Chapter 9.3.3.2]

Keyword-driven scripting

A more sophisticated way of performing data-driven testing is keyword-driven scripting. The idea in keyword-driven scripting is to define higher-level actions that are more readable than singular commands. What were data files for data-driven testing may be called 'action word files' and there is no longer a need for multiple controls scripts. [9, pp. 34-35]

The keywords in keyword-driven scripting are close to services or procedures in programming: they provide the ability to perform the same actions with less work and with different parameter values. Therefore, the keywords are often used to represent use cases or business interactions on a higher level. Example keywords given by Jonassen include keywords such as "create customer", "find customer" and "edit customer" and the syllabus gives similar examples: "create account", "place order" and "check order status" [9, pp. 34-35, 25].

There are numerous advantages to keyword-driven scripting. According to Jonassen, the advantages are the clearest to those who control the execution, especially if they are not technically sophisticated. The advantages mentioned by Jonassen include the facts that keywords can be chosen to reflect business actions, the test execution does not require a lot of technical knowledge, the fact that the keyword-implementation is not dictated by the underlying scripts so that the same implementation can be used for multiple platforms and the robustness of keyword-based testing to minor changes in the SUT. [25]

The ISTQB syllabus is very much on common ground with Jonassen here but has perhaps more emphasis on the scalability of testing once the keywords have been defined. Also, much like Jonassen, the syllabus deems the readability and the possibility to communicate about tests in terms of high abstraction level actions as major advantages in comparison to the data-driven approach. [9, p. 35]

One of the downsides of keyword-driven testing is that having a good overview of the test assignment, which is required, can be demanding [25]. Keyword-driven scripting also means that there are several layers between the test executor and the SUT, which means that maintaining the integrity of these layers is necessary [25, Chapter 9.3.3.6]. As it requires a lot of effort to implement the keywords, keyword-driven scripting may be too costly for smaller systems [9, p. 35]. Besides, to take advantage of the keywords they need to be well defined, poorly designed keywords may only be used once [9, p. 35].

Jonassen and the syllabus both point out that when using keyword-driven scripting, the actual low-level scripts still need to be written at least once and maintained as with other automation techniques [9, p. 35, 25].

Process-driven scripting

Process-driven scripting is yet another step to more advanced test automation. Process-driven scripting is much like keyword-driven scripting but with scenarios that represent uses cases of the SUT constituting the scripts. The scripts are also parametrized or combined into test definitions on a higher level. The idea of these types of definitions is that dealing with actions becomes easier as the logical relationship between different actions can be determined. [9, p. 35]

The main advantage of process-driven scripting is that test procedures can be defined with the workflow in mind, preferably utilizing libraries in doing so. The main downside in process-driven scripting is that sometimes the processes of the SUT, and therefore also the process-oriented scripts, are difficult to grasp. In addition, similarly to keyword-driven scripting, the process implementation has to be done carefully to avoid the use of poor keywords and processes. [9, p. 35]

Behavior-driven testing

Behavior-driven testing is a fundamentally different approach from the ones presented above, as those are mainly used for regression testing. Behavior-driven testing is related to behavior-driven development (BDD), which, according to Agile Alliance glossary, combines TDD and ATDD [1]. In essence, this means that first tests are designed to test the behavior required by the system and the system is built on those grounds so that the tests pass. According to [14, pp. 45-46], behavior-driven testing is the third approach of testing, along with keyword-driven and data-driven testing, that the Robot Framework mainly supports. However, behavior-driven testing differs from the other approaches very much in principle and this study is focused on regression testing so it will not be analysed more deeply here.

4.1.3 Tools and techniques

There are several tools categories that are used for test automation as the idea of tools is to attempt to automate as much of the test automation as possible. The tools are not only related to test case execution. Examples of other tools include categories such as test management tools, which are used in the assistance of test management tasks, for example, as well as scheduling and documentation and fault-injection tools, which are used to intentionally create defects in the SUT for use in defect-based testing. Performance

testing tools, which are used to produce heavy data loads on a product or measure its performance, are another example [25]. Bandeira et al. [10, p. 86] also describe a number of test support tools and place test management tools as a subcategory under those. Other test-support tools include, for instance, metrics reporting tools that give information on which parts of the code and how well it has been tested and usability-measurement tools that may be used as assistance in human-performed verification [10, pp. 86-87]. However, most of these tools are not essential for the use of this study and therefore not described in depth.

An important set of tools are the comparison tools that are used along with test oracles to compare expected results to the actual realized results. This can be done in numerous ways. For instance, a tool may compare files, differences in texts, bitmaps or positions. Tools like these provide the ability to verify a lot of data tirelessly and fast. On the other hand, they tend to produce a large amount of irrelevant data on top of little useful data. Therefore, according to Jonassen, these tools should also enable filtering of some of the outputs. [25, Chapter 9.3.3.7].

4.1.4 Test oracles

The ISTQB glossary [33] defines a test oracle as a "source to determine expected results to compare with the actual result of the system under test" and a survey on test oracles [11, p. 507] as "a procedure that distinguishes between the correct and incorrect behaviors of the System Under Test". Jonassen defines test oracle as "a special concept in test automation" that "is used to determine expected results from inputs" [25, Chapter 9.3.3.3]. Furthermore, she goes on to describe Automated test oracles as "tools that can generate the expected results" for identified inputs [25]. A good example of an easy to find oracle is an old system that is going to be replaced. The old system can be used as an oracle if the new one is supposed to have the same functionality [25]. In this case, the old system can be used to provide the expected result for the new system by giving the old system the same inputs as the new one is given [25].

It seems like the selection of the test oracle is one of the important questions when implementing test automation because of the cost-benefit assessments that are related to each one of the options. The ISTQB syllabus for TAEs mentions the necessity of using automated test oracles as one of the limitations of test automation because test automation is only able to check those results that could be verified by an automated test oracle [9, p. 13]. Similarly, the oracle problem survey [11, p. 507] states that none of the test oracle techniques for automation presented in the literature are completely adequate and that all forms of test oracles, including human supervision, cause challenges in cost reduction and benefit increases. Jonassen's book about software testing is on common ground with the previous two on this as it mentions in [25, Chapter 9.3.3.3] that some say that the tester himself is the best test oracle but remarks that the tester being the oracle is not always possible. The book [25] also mentions risks involved with test oracles such

as the possibility of a false sense of reliability, the possibility of not achieving sufficient coverage and the possibility of repetition of faults between the test oracle and the SUT. For example, the ISTQB syllabus gives invalid test oracle use as a possible cause of an often dangerous false-pass situation in which a defect in the SUT goes unnoticed by the test automation [9, p. 55].

The oracle problem survey divides solutions to the oracle problem into four categories: specified, derived and implicit test oracles, as well as solutions in which oracle automation is not possible but human effort can be reduced. Specified oracles are defined with (preferably mathematical) specification logic, written in a specification language [11, p. 512]. Derived test oracles rely on artifacts, for example, documentation, different versions of the SUT, or properties of the SUT in distinguishing the correct behavior from incorrect [11, p. 514]. An implicit test oracle draws its conclusion on whether the results are correct implicitly from general knowledge [11, p. 518]. A different way from the previously mentioned ones is the human oracle problem solution, in which the aim is the reduction of the Human Oracle Cost [44] by, for example, providing guidance tools or assisting humans on what to focus on when testing. [11, p. 519]

4.2 The implemented scripted test automation

The traditional, keyword-driven scripting test automation technique implemented in this study was done using Robot Framework [47]. Robot Framework is an open-source framework for implementing hierarchical acceptance test automation originally developed at Nokia Networks. With the help of varying libraries, it is usable for several purposes varying from web-application testing to desktop and mobile application testing. The extensible nature of Robot Framework provides versatility and flexibility for test automation in different environments and as well as in the scope of a single project [14, pp. 9-10].

According to Sumit Bisht's book named Robot Framework test automation, the Robot Framework supports three approaches to creating scripts: keyword-driven, data-driven, and behavior-driven tests [14, pp. 45-46]. The way the test automation was implemented for this study is closest to keyword-driven testing, with some elements of data-driven testing included as in the use of different parameters with the same keywords for performing the same actions with different inputs, expecting different outputs. The following subsections explain how the implementation of keyword-driven test scripts was done in this study, how the scripts were controlled, what dependencies were necessary in doing that and how the result verification was done.

4.2.1 Variables

Variables are defined in a resource file. Program 4.1. is a snippet from the resource file that defines variables linking them to UI elements. On line 1 in Program 4.1. the variables

table is started and variables named SPEAKER_TEST, SPEAKER_Q, PASS_BUTTON, and FAIL_BUTTON are defined. In Robot Framework, instances of different concepts such as variables and keywords are defined in tables. The start of the scope of the variables table is indicated on line 1 of Program 4.1. The scopes of other tables are indicated similarly and a scope continues until another one is begun.

```
1 *** Variables ***
2 ${SPEAKER_TEST}    xpath=//android.widged.TextView[contains(@text,'Speaker')]
3 ${SPEAKER_Q}       TextView[contains(@text,'Is the speaker operating as expected?')]
4 ${PASS_BUTTON}     ImageButton[contains(@resource-id,'btn_image_pass')]
5 ${FAIL_BUTTON}     ImageButton[contains(@resource-id,'btn_image_fail')]
```

Program 4.1. *The resource definition, connecting Android UI elements to variables.*

The variables defined in Program 4.1 are matched with Android UI elements by identifying their type and some attribute of the element that is desired. For example, the variable SPEAKER_TEST is linked to the Android textview-type widget that contains the text "Speaker". This text view is visible at the top of the first two screenshots in Figure 3.1 on page 14. Other attributes, such as resource-id or placement constraints, can also be used in identifying the elements. The beginning of Android widget XPath definition is omitted on lines 3-5 to save space and make the snippet more readable.

4.2.2 Keywords

Robot framework test automation uses keywords in scripts. There are several basic keywords and keywords provided by libraries, but a user can also define higher-level keywords to avoid repetition and make tests more readable.

The user-defined keywords are defined in the keyword-table. The keywords are defined inside the table by indicating the name of a keyword on one line and the actions to take when the keyword is called on the following lines with indentations.

In Program 4.2, two higher-level keywords are defined using variables and some basic keywords. On line 3, the UI element with the variable SPEAKER_TEST is waited for, for the default time of five seconds. On the two following lines, the visibility of two other UI elements is checked without waiting for them using the Page Should Contain Element keyword. On line 6, the pass button for the speaker test is tapped. The keyword continues with similar checks and a fail button tap for the second part of the speaker test. From line 13 onwards, the keyword "Fail Phone Speaker Test" is defined. This example code does not contain all of the checks performed during the actual test automation, some of them have been left out.

All of the variables that are defined in Program 4.1 are used in the keywords defined in Program 4.2 in waiting for, checking the existence or tapping UI elements. By calling these keywords in order of appearance, the three screenshots that are presented in Figure 3.1 on page 14 are checked and the buttons highlighted in the figure are tapped.

```
1  *** Keywords ***
2  Pass Fail Speaker Test
3      Wait Until Element Is Visible        ${SPEAKER_TEST}
4      Page Should Contain Element          ${SPEAKER_RIGHT_TEXT}
5      Page Should Contain Element          ${SPEAKER_Q}
6      Tap                                  ${PASS_BUTTON}
7
8      Wait Until Element Is Visible        ${SPEAKER_LEFT_TEXT}
9      Page Should Contain Element          ${SPEAKER_TEST}
10     Page Should Contain Element          ${SPEAKER_Q}
11     Tap                                  ${FAIL_BUTTON}
12
13 Fail Phone Speaker Test
14     Wait Until Element Is Visible        ${PHONE_SPEAKER_TEST}
15     Page Should Contain Element          ${PHONE_SPEAKER_QUESTION_TEXT}
16     Page Should Contain Element          ${PHONE_SPEAKER_INST_TEXT}
17     Tap                                  ${FAIL_BUTTON}
```

Program 4.2. *Keyword definitions for passing and failing left and right speaker test and failing phone speaker test.*

4.2.3 Example of a test case

Test cases in Robot Framework -based test automation are defined using keywords and variables. Program 4.3 is an example of a test case that runs the audio test set and checks the results. The words that start the scope of the test case are also the name of the test case, "Run Audio Set And Check Results" in this case. As with keyword definitions, the line that contains the name is not indented. The following lines are indented and contain the keywords defining the actions to be taken.

```
1  *** Settings ***
2  # Contains environment setup (omitted)
3
4  *** Keywords ***
5  Send Test link
6      Run Process ../Scripts/send_link.sh ${DEVICE_ID}
   ${AUDIO_LINK}
7
8  *** Test Cases ***
9  Run Audio Set And Check Results
10     Pass Fail Speaker Test
11     Fail Phone Speaker Test
12     Check Microphone Help Page
13     Fail Microphone Test
14     Pass Record And Play Test
15
16     Open Failed Audio Set Results
17     Check Record And Play No Issues
18     Check Speaker Passed With Issues Left Issue
19     Check Phone Speaker Issue
20     Check Microphone Issue
```

Program 4.3. *An example of a test case for audio test set. The test case name is on line nine and the following lines each invoke a keyword.*

The high-level actions that are taken by the test case in Program 4.3 are defined in plain English in the names of the keywords that are run. The keywords that are defined in Program 4.2 are used on lines 10 and 11. In short, the audio test set is executed in pretty much the same way as in Figures 3.1 on page 14 and 3.2 on page 15, after which the results are checked as in Figures 3.3 on page 16 and 3.4 on page 17. In addition to what is seen in the figures, the record and play test is passed and the issues are checked for phone speaker and microphone tests of PiceaOne. Also, in addition to simply tapping the buttons, in the implemented test automation checks are performed on as many of the UI elements as considered necessary. It is noteworthy that also the basic keywords and parameters can be, and most often are, used when defining test cases.

4.2.4 Test execution tools

The execution of Robot Framework tests on Android devices requires additional tools and libraries to be used in between Robot Framework and the SUT. Robot Framework itself runs on Python so installing python is naturally a necessity. In addition, the AppiumLibrary [5] for Robot Framework has to be installed in order to make Robot Framework work correctly with Appium. The basic, predefined keywords such as "Wait Until Element Is Visible" and "Tap" that were used in this study were most often AppiumLibrary keywords.

Appium [4], an open-source test automation framework for mobile apps was used as the core tool for running tests on the mobile device. Appium provides a client-server architecture for communication between the tool running the tests (Robot Framework in

this case) and the mobile device. Appium requires Node to function so installing Node is also necessary.

Finally, the Android SDK is necessary to communicate between the test automation computer and the mobile device. The correct path to java and the Android SDK needs to be defined in system paths. The test execution requires an Appium server to be running. The information on the device to be connected to and the server to be used for the connection is included in the resource files. Standard keywords may be used to establish a connection, open the application and close the application after testing.

4.2.5 Test Oracle and comparison tools

The test automation produced for this study is regression testing and regression testing relies on the assumption that previous versions of the SUT can be used in determining whether a result is correct or not. This means that the oracle falls into the category of derived oracles [11, p. 507]: the expected UI elements are all predefined and their order and appearance are expected in accordance to what happened when the same actions were performed on an older version of the system.

Program 4.1 on page 28 is a good hint of what the comparison tool consists of. With AppiumLibrary, each element that needs to be found has to be identified by either a locator or a webelement [5]. A webelement contains an instance of a WebElement, which represent elements on the UI and a locator is a string specifying how to find a certain element of the visible ones [5]. In the test automation implemented in this study, locators are used to define variables of UI elements as they are in Program 4.1. When a keyword such as "Wait Until Element Is Visible" is called with a parameter and an optional parameter for wait duration, an UI element with the locator defined in the parameter is waited for. If it appears before the wait duration is reached, that keyword assertion passes and execution continues normally, and if it does not appear in the given time, the test case fails, and execution moves to the next test case.

4.2.6 Technique evaluation

Robot Framework is a useful tool for creating keyword-driven test scripts with high-level keywords and variables. This makes test script creation more efficient than scripting purely with a scripting language. Using the framework is intuitive and does not necessarily require advanced programming skills, especially after the definition of the keywords as they can be named according to the actions they perform. Robot Framework and the additional libraries also enable the creation of test scripts for different devices for both Android and iOS platform. Another advantage is that getting the test automation started is fast and requires little effort.

A major pitfall of Robot Framework, at least of the ones noticed in this study, is the fact that Robot Framework uses a lot of dependencies in communication with the SUT. This causes extra fragility because of various errors occurring in the tools or between the tools. For example, Appium timeout failures and socket connection problems were fairly common during the time testing performed in this study. Also, the detection of certain types of defects in the UI is not possible as the elements are defined using their ID or the texts in the elements which can cause serious false-pass situations. A better example of the failure in detecting a defect like this is presented in Section 6.3.

A third noticeable downside of manual scripting is the necessity of plenty of programming effort in writing scripts for comprehensive test scripts for a complicated application. Even though keyword-driven scripting and Robot Framework surely have advantages in terms of efficiency in comparison to some frameworks and tools with less abstraction and hierarchy, it is still necessary to manually write each test case, each test path, and each PiceaOne test run combination separately. For example, when writing a test case that runs every one of PiceaOne's test cases, even though high-level keywords with hierarchy could be made to ease the effort, there was be a lot of repetition of code. Even though some of the test scripts in this study were created using high-level user-defined keywords, a comprehensive test suite would require a lot of work. To top it off, if one wanted to create ten test cases that each run a full PiceaOne operation with all of the test cases five times, retrying after each of the runs and executing each of the runs with different PiceaOne test case results, it would understandably require a great deal of effort if done with the technique of scripting that was presented in Subsections 4.2.1-4.2.3.

To summarize, the Robot Framework test automation solution implemented in this study has the characteristics of a typical keyword-driven manually scripted test automation. Some of those characteristics were mentioned in Subsection 4.1.2. The implementation is fast and easy to start with, the keywords and variables enable the creation of highly abstract test cases and reuse of code, which in turn provides efficiency in writing more comprehensive tests. Some of the pitfalls mentioned in Subsection 4.1.2 are noticeable: while the framework is flexible, it also requires several layers between the test cases and the SUT, causing some reliability issues. The poor or inadequate design of keywords also caused some issues in this study. The tests that were implemented could have been more numerous and implemented with less effort if user-defined keywords had been used from the beginning.

5 MODEL-BASED TESTING

According to ISTQB, model-based testing is "testing based on a model" [34]. Thus, it can probably be said that model-based test automation is test automation based on a model. Perhaps one of the reasons the ISTQB definition sounds somewhat reduced is mentioned in Model-Based Testing Essentials by A. Kramer and B. Legeard [13, Chapter 2.1] where the point is made that a model does not necessarily describe the SUT but may describe, for example, the test itself.

Even though MBT is fundamentally different from the techniques mentioned in Chapter 4, it is noteworthy that according to the ISTQB definition MBT still falls into the category of scripted testing [35]. The major difference between traditional testing and MBT is that in MBT the scripts can be generated by a tool. According to M. Utting and B. Legeard [49, p. 26], the automated design and generation of test cases are two of the three main issues MBT aims to solve, the reduction of maintenance costs being the third. Kramer and Legeard do not describe MBT as a paradigm-shift but still call it a clear change, an extension of a sort, to "traditional testing" [13, Chapter 1.4].

This chapter deals with the theory of MBT and test automation with an emphasis on the models, the notations for creating models and the strategies of creating test cases. The model-based approach to testing and its advantages and downsides are explained based on three main sources [13, 38, 49]. The implementation of model-based test automation that was used in this study is then presented and analyzed. While MBT principles can be used to manually generate test cases for use in manual testing, in this study model-based testing refers to model-based test automation.

5.1 Model-Based Testing theory

Unlike with the scripting techniques that were discussed in Section 4.1, test cases are not manually scripted by a person in automated MBT, rather they are automatically generated by a tool based on a model of the SUT [9, p. 26, 49, p. 8]. Model-based test generation can be used to derive tests for any of the approaches presented in both Section 4.1 and the syllabus for advanced level test automation engineers [9, pp. 31-36]. It should be noted that the syllabus does not mention behaviour driven testing. Paul Jorgensen divides MBT to three types based on the level of automation [38, p. 8]. In the first type, a model is used for generating abstract test cases to be used as wished [38, p. 8]. A step

toward more automation from this is the case where abstract test cases are first selected for execution according to some criteria and turned from abstract cases to executable ones. With the highest level of automation, the test cases are also executed on the SUT [38, p. 8].

MBT is mentioned in Jonassen's tools-focused approach of test automation in that test design tools use models of the SUT in the generation of the test cases [25, Chapter 9.3.3.1]. In addition to the ability of test design tools to derive and generate test cases based on specifications, she mentions that they can generate input for test cases and even derive good quality test cases based on the actual source code [25, Chapter 9.3.3.1]. The generation of input for test cases can also be based on models, input models in this case [25, Chapter 9.3.3.1], and it seems to be regarded a part of MBT [49, p. 7].

5.1.1 Models

As eloquently described by P. Jorgensen in his book The Craft of Model-Based Testing [38, p. 3], a model is considered to express the stimuli and the responses of the SUT, i.e., the inputs that are given and outputs that are received. Jorgensen divides design models into two general types, structural and behavioral models. He mentions that structural models focus on classes, attributes, methods and the connections between classes. He then goes on to list nine different behavioral models, analyses them and the extent to which they support MBT [38, p. 7]. According to Jorgensen, Harel statecharts and activity diagrams are the two main examples of behavioral models [38, p. 7].

Utting and Legeard list four different modeling approaches: generation of input data from a domain model, generation of test cases from an environment model, generation of test cases with oracles from a behavior model and generation of test scripts from abstract tests [49, p. 7]. In the generation of input data, the model is a representation of the usable data domain and the generation provides a subset of that to be given to the system. This can be useful but takes no stance on whether a test case was passed or failed. Similarly, environment model-based generation does not specify expected outputs, it rather just provides the ability to make calls to the SUT. This is due to the fact that environment models do not model the behavior of the actual systems but only their environment. Therefore, Utting and Legeard highlight the third generation method, generation of test cases with oracles from a behavior model, as the most sophisticated and complex one. This method generates input values and sequence calls, and it encompasses the oracles that can be used to verify correct output from the system. The model used in this study is a behavioral model in that sense. With the fourth approach, the generation of test scripts from abstract tests, Utting and Legeard refer to generating low-level tests scripts from what Jorgensen [38, p. 7] categorizes as structural models. [49, pp. 7-8]

An important question when it comes to modeling a system is how accurate the model should be [49, p. 9]. Utting and Legeard [49] approach this issue by taking a look at two

definitions of the word model. These slightly contradictory definitions go as follows: "a small object, usually built to scale, that represents in detail another, often larger object" and "a schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics" [49, p. 9]. As mentioned by Utting and Legeard [49, p. 9], the desired model size is smaller than the size of the actual SUT so that the cost of creating a model makes sense, but also detailed enough to imitate all the characteristics of the SUT that are to be tested. Jorgensen describes this issue as aiming for a model that is necessary and sufficient, not too weak nor too strong [37, Chapter 22.2]. In other words, when creating the model, an engineer should think of an appropriate level of abstraction, keeping in mind that most often the goal is not to specify everything about the SUT [49, p. 60].

On top of the level of abstraction in the model on an operational level, that is, leaving some features of the SUT out to simplify the model, Utting and Legeard mention plenty of other important things to consider when modeling a system. For example, the same abstraction principle should be considered for each of the input and output parameters. The recommendation is that if one does not want to test the effect of a parameter on the system, it is better to leave it out than include it in the model. Another suggestion is wrapping multiple SUT operations inside one model operation when it feels logical, and vice versa. Finally, after a model has been implemented, it is recommended to validate and verify the model. Validating and verifying means checking that the model depicts the desired behavior and that it is consistent and correct. Some tools can be very useful with this. For example, a tool may offer syntax checking or model validation by simulation. [49, pp. 60-61]

5.1.2 Notations

There is a multitude of possible notations to be used for modeling the behavior of a system. Utting and Legeard present seven paradigms: pre/post, transition-based, history-based, functional, operational, statistical and data-flow notations. Transition-based notations focus on the transitions between states, data-flow concentrate on the flow of data through the SUT. Statistical notations depict systems as probabilistic models of events and inputs, operational notations as collections of executable processes, and functional notations as collections of mathematical functions. History-based notations describe systems using traces of its allowable behavior over different periods. State-based notations depict systems as groups of states. [49, pp. 62-64]

Notation based on preconditions and postconditions, also known as state-based notation and sometimes also just called "model-based" notation, is used for modeling in this study. State-based notations model a system as a group of states based on the values of variables. Together the variables form a snapshot of the state of the system. Operations that modify the state-defining variables are used to change states. State-based notations usually define an action as a pair of a precondition and a postcondition. [49, p. 62]

The state-based and transition-based notation are the most popular modeling notations. The selection of a notation that fits the purpose is not trivial and depends very much on the SUT characteristics. According to Utting and Legeard, state-based notations are best for data-oriented systems whereas control-oriented systems are recommended to be modeled using transition-based notation. Wrong notation selection is likely to make modeling more difficult but few systems are fully classifiable to one or another orientation in terms of data and control and the modeling of one type of system with a notation preferred for the other is possible. A more essential restrictive factor is that the notation has to be formal, meaning precise and unambiguous. This is imperative especially in performing simulation on the model and for the model to be usab as an oracle. [49, pp. 64-65]

While Utting and Legeard mainly focus on notation comparison regarding making modeling choices based on the nature of the system [49, pp. 64-65], Jorgensen highlights the choice of model as the factor determining the ultimate success of MBT [37, Chapter 22.2]. He mentions the expressive power of models, the system's nature and the analyst's capability to use different models as essential aspects in choosing the model [37, Chapter 22.2].

Based on the nine behavioral models mentioned by Jorgensen [38, p. 7], the state-based notation is probably the most compatible with statecharts. The statechart is a hierarchical directed graph that encompasses states, transitions between states, orthogonal regions, inputs, and outputs. A statechart state may contain substates meaning divisions or variants of broader states, and orthogonal regions meaning similar, parallel regions within a state. Supporting the claim of similarity between the state-based, i.e. pre and postcondition-based notation and statecharts, it is mentioned in The Craft of Model-Based Testing [38, p. 148] that inputs and outputs "can be any combination of events, data values or conditions involving any of these". Inputs cause and outputs are caused by transitions in statecharts. [38, pp. 147-166]

Notably, the SUT of this study could be explained somewhat logically with a statechart. A PiceaOne test case could be interpreted as a substate and the encompassing test set as a state containing this substate. Also, depending on whether a PiceaOne test case is failed or passed the execution could be interpreted to be taking a different route as in an orthogonal state, as most of what follows does not depend on the result of a single test case.

5.1.3 Strategies

The ISTQB definition for a model-based test strategy is that it is "a test strategy whereby the test team derives testware from models" [36]. Test strategies can be used in various ways. For example, Bae et al. [8, p. 42] give an example of using test strategies to first create short test cases for scanning shallow states of the system quickly and later

creating longer cases that may cover a wide range of behavior using a dynamic event-extraction based technique. The advanced level syllabus for test automation engineers [9, p. 28] brings up the choice of test generation strategies, for example, state or transition coverage for behavior-based approaches, and test selection strategies that may be used to define coverage criteria, weighting on certain areas or risk assessment. The test selection strategy can also be useful when facing a possible test case explosion, a problem that is better explained in Subsection 5.2.6 [9, p. 28].

The Craft of Model-Based Testing presents a case example of a model-based on a system that controls a garage door. The tool used for test generation detects that the model implements a finite state machine and recommends four applicable strategies based on an automatic analysis on a reference of the model. *Basic control state test cases* strategy creates a suite with the minimum number of cases for visiting each state of the model and *transition test cases* strategy creates a suite that has test cases for visiting each transition. *Requirements coverage* strategy allows the creation of selective test suites that can be generated by expanding the model to include a list of requirements linked to elements of the reference model. These requirements could then specify what to have a focus on. Full requirement coverage is reached when all of the selected requirements are tested by at least one test case [13, Chapter 8.1]. The last strategy, *conformance testing* strategy, is the most comprehensive. To use this strategy, the user of the tool needs to indicate a fault hypothesis, a statement on the number of internal states the state machine may have. Assuming the given number was correct, every deviation from normal or unexpected behavior by the SUT will be detected by the conformance strategy generated test cases. The conformance testing strategy was selected in that case because of the safety-critical nature of a garage door. [38, pp. 388-389]

Practical Model-Based Testing by Utting and Legeard [49] describes a lot of *coverage criteria* which it juxtaposes with the strategies on pages 132, 209 and 265, for instance, by stating that "an easy strategy would be to use all-transition-pairs coverage" [49, p. 265]. So, in essence, the use of a coverage criterion is a strategy in this case. The mentioned coverage criteria are plentiful, including coverage criteria such as all-states, all-configurations, all-transitions, all-transition-pairs, all-loop-free-paths, all-one-loop-paths, and all-paths coverage criteria. The meaning of the first four relates to covering all of the states, configurations, transitions or transition-pairs at least once. The following coverage criteria mean covering each non-looping path at least once, covering each at most two repetitions of one configuration path at least once and covering each path at least once. [49, pp. 117-119]

To summarize, strategies are a way to better affect various elements of MBT. They can be used to define what kind of test suites are to be generated and therefore, what kind of tests are going to run, which areas are to be focused, how deep or shallow testing should be, what kind of paths to favor in testing etc. Strategies can be used to execute tests best fit for the type of testing in question, for example, requirements coverage for regression testing [38, p. 389].

5.2 The implemented model-based testing solution

The model-based test automation solution in this study uses the fMBT-testing tool [22] by Intel and it operates on the highest level of automation of the ones mentioned in Section 5.1. The model is created using the AAL/Python notation language which uses pre/postcondition notation. Even though fMBT had caused some difficulties with setup, programming and visualizations in the case study mentioned in The Craft of Model-Based Testing [38, pp. 421-422], the setup and running of simple tests for an Android device with the instructions [24, 21] and tutorials [23] provided in the spring of 2019 did not feel that difficult. Also, the visualizations presented by the fmbt-editor proved quite useful.

5.2.1 Notation

As mentioned, fMBT uses preconditions and postconditions in the model definition. Even though one could argue that PiceaOne Diagnostics is more of a control-oriented than a data-oriented system which would imply a transition-based notation, the pre/postcondition approach worked well for modeling it. The model was implemented pretty much the way Utting and Legeard suggest for using pre/post notation on a node-based system [49, p. 65]: state variables were used to indicate which state the system was in and the conditions were also defined using these variables. This leads to the necessity of operations also changing the values of the state variables to indicate changes in state.

The state variable values are continuously changed when progressing with a test run. This happens in either inputs or outputs. Inputs are definitions of possible actions that can be taken if a set of conditions is met. These conditions are based on variable values and checked in AAL/Python in the guard blocks. The inputs, also known as test steps, are triggered by the test generator. Their effect on the variables is defined in the body block where the variable values are changed and hence the state also. Adapter blocks define the actual actions to be taken in the system that is being tested. In this study, that would be button taps and UI observation, for example.

Program 5.1 shows the definition of *start audio set* input, an action to start PiceaOne and start running the audio test set. The guard block defines the precondition that the system must be in a state where the home screen is shown using the *fg_app* variable. The adapter block uses a shell command via ADB to order the activity manager to start the PiceaOne beta application with the link that defines the tests to be run on PiceaOne, and then waits for a second. In the body block, five variable values are changed to indicate to the system that a test run is ongoing and more precisely, that the audio test set and the speaker test case is being run. Besides, the variable for foreground application is changed from the home screen to PiceaOne and the "audio" key is added to a dictionary which will be used to save the results of PiceaOne test cases. The shell command on line 4 of Program 5.1 has been turned to pseudo-code.

```
1     input "start audio set" {
2         guard{ return fg_app == "homescreen" }
3         adapter() {
4             device.shell("am start piceaonebeta")
5             waitOneSecondTransition()
6         }
7         body() {
8             ongoing_test_case = "speaker"
9             fg_app = "piceaone"
10            piceaone_run_ongoing = "True"
11            audio_run_ongoing = "True"
12            results_dict["audio"] = {}
13        }
14    }
```

Program 5.1. *Example code of an input that starts PiceaOne run with audio test set.*

Actions can also be taken based on observations made on the SUT. In that case, the test steps are called outputs. These observations are made in adapter blocks of outputs, and if they return true, the step is performed. The generator validates whether it was allowed to take the test step and if so, the guard block returns true, and the body block of the output is executed. If what was observed was not allowed, the test fails. However, outputs were not used in this study as the simple use of inputs, if possible, is recommended in the fMBT tutorial [23]. The tutorial states that tests that consist of just inputs and verifications are "simple, reproducible, deterministic, and easy to debug" [23]. [24]

Inputs and outputs may be parts of a bigger collection of states which can be labeled by tags in AAL/Python. A tag can be used to perform checks for multiple inputs in the tag's guard block and verifying system functionality in the adapter block without having to repeat code. Tags are also useful in making the model visualization easier to grasp.

In addition to tags, inputs, and outputs, the AAL syntax includes sections such as variables, for defining the state variables, and language, for defining the language to be used, Python in this case. Initial state block defines the state at which the testing starts, and there are adapter blocks for execution before and after test runs.

5.2.2 Model

In the model of PiceaOne Diagnostics, the initial state was defined to be a state in which no tests on PiceaOne are being executed and the device on which tests are run is showing the home screen with the PiceaOne icon visible. From the initial state, there are 14 possible ways to continue: starting each of the 12 test sets separately, merely opening PiceaOne to check version information and QR code reader, and starting a test run that includes all the test sets. The state of a PiceaOne test set being executed is defined in a state, indicated by a tag such as the *audio-run-ongoing* tag, which includes all the actions

```
1 tag "audio-run-ongoing" {
2     guard() {return audio_run_ongoing == "True"}
3     adapter() {}
4     input "cancel audio set" {}
5     input "check audio help pages" {}
6     input "pass speaker" {}
7     input "pass fail speaker" {}
8     # A lot of inputs omitted
9 }
```

Program 5.2. *The beginning of audio-run-ongoing tag.*

to be taken while the audio test set is being run. The beginning of the implementation of this tag is shown in Program 5.2. Some variables and code have been omitted from the program because the idea of the piece of code is just to indicate what a tag like this consists of.

As the inputs in Program 5.2 imply, there is an input for each way a test case can be executed. The cancellation and help page checking inputs work for any test set, and there are four inputs for speaker test because there are four ways to execute it on top of canceling, as specified in Chapter 3. Some PiceaOne test cases are executed simultaneously, and for those, there are no separate inputs but they are usually handled as a group similar to how the individual cases are handled. Therefore, one could say the modeling in detail is based also on the view that is being shown in addition to the test case that is ongoing. This, and the decision that the inputs were not more detailed than this was based on the kind of abstraction level decisions discussed in Section 5.1.

Figure 5.1 on page 41 is a screenshot taken of the fmbt-editor tool showing the first lines of modeling code on the left and a visualization of the model created in this study on the right. Only the audio test set part of the model is included. The visualization has been edited to show the names of the inputs in green in a bigger font. The thin green arrows indicate inputs, and the thicker green arrows are there just to point out that the cancel input creates four separate actions/arrows on the visualization. On the right, the names of the states that are indicated by the boxes are explained and numbered. These states are most usually situations in which a test case has been started, and some input is to be selected. State number seven, PiceaOne home, leads back to the initial state, so the model is not finite like that. These states are also presented as a point of reference in Figure 5.2 on page 42.

The texts in the boxes list the variables that affect the visualization complexity and their values. Also, the input *check audio help pages* is written inside each of the boxes because it does not change any of the state variables and therefore there is no need for an arrow to another state. This input is used only to check out the help pages like the one shown in the rightmost screenshot of Figure 3.2 on page 15. The adapter block of the input uses the variable *ongoing_test_case* to determine what to expect from the opened help page. Returning from the help page can be done in four ways, as is highlighted in the above

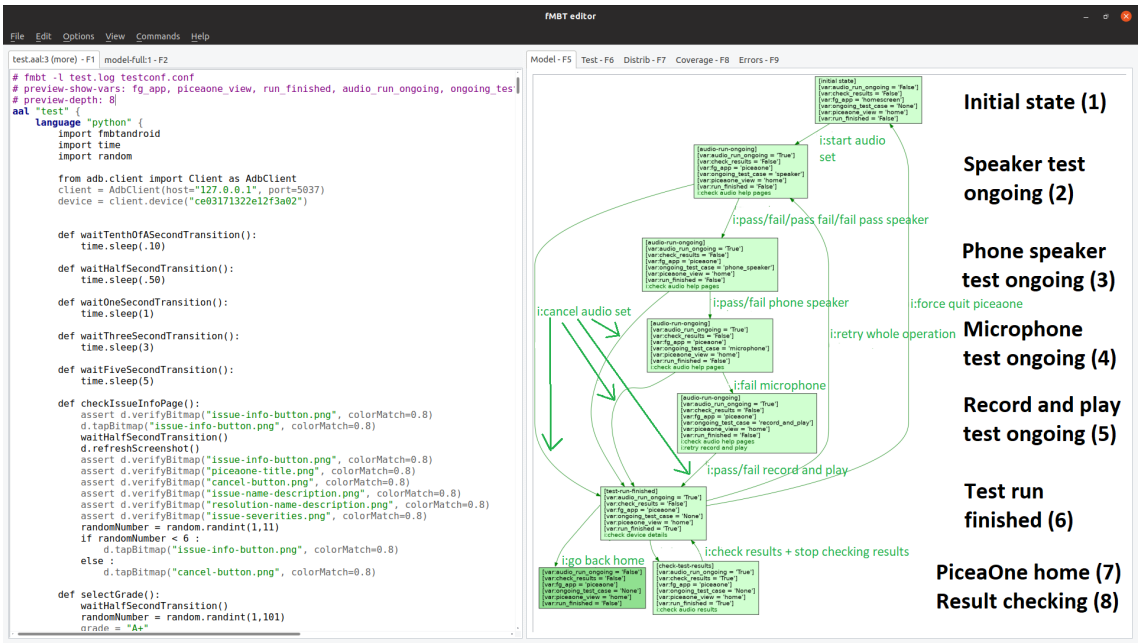

Figure 5.1. *A picture of the model for audio test set testing, without including PiceaOne test results in the variables with effect on the list.*

mentioned screenshot. The decision on which way to go back is done using Python's random-library. This is another example of a way to simplify the model without having to compromise a lot of what is to be tested. Random-library was used in other instances too, for example in selecting grade which could vary from A+ to C- and not available. Creating separate inputs for selecting each of the grades felt like going too high in detail.

To control the areas to be tested, strategies and different configurations for the model were used. Each of the tags and the inputs resulting in conditions that lead to these tags were defined in separate files. Inclusion of these files was controlled within configuration files that also contained the definition of what kind of strategy to use, how many tests steps are to be executed, what to do on failure, pass, and so on. The strategies that were used will be explained in the next subsection.

Generating and executing different types of tests based on different configuration files was done in a test script. The most commonly used configurations included either just one or all of the files defining the tags. In case all of the files were included, the whole model was tested in the test suites generated based on those configurations. The fmbt-editor visualization of the whole model is shown in Figure 5.2, where each of the boxes is a state, just like in Figure 5.1. The audio test set part of the model is highlighted with the boxes with red borders, and the numbering in the boxes matches the numbering of the states in Figure 5.1. Note that the full run branch (leftmost in the figure) of the model, which would contain all of the 12 test sets executed in a row, is cut short. This had to be done because the fairly large size of the model was already causing processing issues.

The branches in Figure 5.2 represent the PiceaOne test sets, each one ending in state 7, the home view. The branches from left to right, beginning in each of the boxes on the sec-

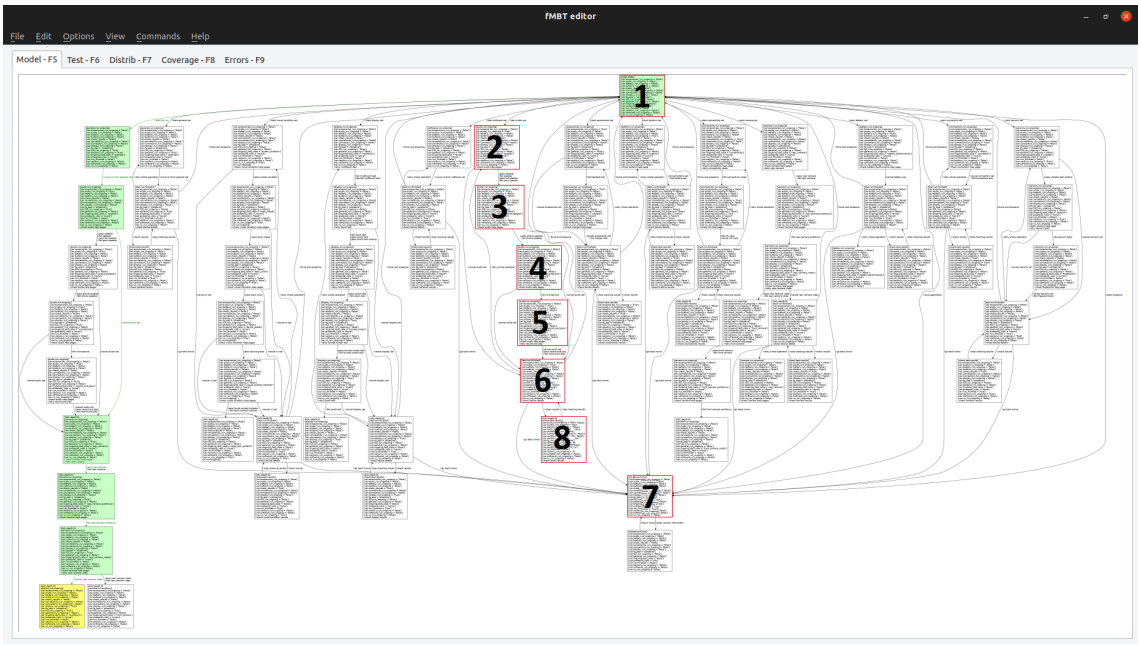

Figure 5.2. *A picture of the whole model of PiceaOne Diagnostics, with the states of the audio test set part of the model that are shown in Figure 5.1 highlighted with red.*

ond row, are full operation with all test sets, general, visual condition, display, software, audio, accessories, buttons, connectivity, camera, battery, connectors, and sensors test sets. It is noticeable that the software and general test sets that do not have user interfaces are shorter in length as there are just a dummy input and state variable changes before moving forward from them.

5.2.3 Used strategies

A lot of the test scripts that were used were randomly generated by fMBT based on the pieces of the model. This was the case especially in the earlier parts of the study when the focus was more on building the model than using it efficiently. Also, it was easier to just split the model into smaller pieces using several configuration files to focus testing on a certain area. In some sense, that could also be called using strategies. By the end of the study, a multitude of the types of strategies mentioned in Subsection 5.1.3 was used for generating the desired types of tests.

At first, runs that focused on retrying, failing, passing, or canceling were defined as strategies that heavily favored the selection of the corresponding inputs. Especially favoring "start full run" input was essential for being able to run full operations more often that individual test set operations, because when all the files including all test set tags were included, also the inputs starting the individual test set runs were included. However, the individual test set runs were often tested briefly before starting the longer runs to know if everything was with basic actions. After all, a failure in the execution of a PiceaOne test set would also cause a failure when testing the full operation.

Strategies were used in various ways. Input keywords such as "pass" and input keyword pairs and combinations such as "retry" and "cancel" were favored or disfavored to run PiceaOne tests in desired ways. The reasons to do this were numerous: focus could be paid on an area of PiceaOne that had just been modified, and on the other hand, certain keywords could be avoided. For example, if it felt necessary to pay attention to testing a long PiceaOne test run to try to find memory issues, the inputs starting with the word "retry" in them could simply be heavily favored, and thus the application was kept on for longer. For a large part of the testing period, cancellations were avoided as there was a defect found that was caused by canceling PiceaOne test sets at certain points of execution. The defect was not high in priority, and so it was not fixed that quickly, but it still often caused failures in test automation. The possibility to avoid cancellations allowed testing to go on for a longer and reduced time in checking failures caused by the same defect. On the other hand, when new features were added or old functionality was modified, testing could easily be focused on a certain test case, for example.

5.2.4 Test generation and execution tools

Before the tests are generated, AAL/Python is compiled into pure Python by the AAL/Python runner remote_pyaal. The compiler that is used in this, fmbt-aalc, is also the one that is used to omitting some part of the modeling files if desired.

Test generation parameters were set in the configuration files one of which was always selected when generating and executing a test run. The main parameters in the configuration files besides model selection are "heuristic" which defines what kind of logic to use in input selection, default being random, "coverage" which determines how coverage is measured lastly the end conditions defining parameters such as "pass", "on_pass", "on_inconc" etc., which define when a run passes, what to do when it passes or what to do in an inconclusive situation, for example. The test cases produced by different configurations and the test steps they included could be viewed using the fmbt-editor. Also, errors, coverage data, and test step distribution among inputs could be observed using the tool.

The fMBT tool was run on a Linux Ubuntu machine. The fmbtandroid python library was used for interaction with the Android system. Android SDK platform tools, ADB in particular, were required dependencies. When using a physical device for testing, it was necessary to enable and allow USB-debugging via device developer options to be able to use ADB-commands.

5.2.5 Test oracle and comparison tools

The oracle in the MBT solution implemented for PiceaOne matches the characterization of a specified oracle by Earl T. Barr et al. [11, p. 512] in The Oracle Problem in Software

```
1  adapter() {
2      d.refreshScreenshot()
3      assert d.verifyBitmap("speaker-title.png", colorMatch=0.8)
4      assert d.verifyBitmap("speaker-right-instruction.png", colorMatch=0.8)
5      assert d.verifyBitmap("speaker-is-it-operating.png", colorMatch=0.8)
6      d.tapBitmap("pass-button.png", colorMatch=0.8)
7  }
```

Program 5.3. *An example of an adapter block beginning that passes the first part of PiceaOne speaker test case.*

Testing: A Survey. AAL/Python also matches the features of a model-based specification language. The language and its notation are necessary for creating the oracle for knowing what is to be shown next if a PiceaOne test case is passed or canceled, but the bigger challenge is the management of the PiceaOne test case results and checking that they are correctly presented on the application.

The test oracle for result checking in PiceaOne is pretty much included in the results_dict variable and how it is handled. The variable is a two-dimensional dictionary that contains PiceaOne test set names on the first level of keys. The PiceaOne test case names are the keys on the second level of the dictionary that connect to values, the results for each of the test cases. These values are added in the body blocks of inputs that determine results for the test cases. For example, when "pass speaker test" input is performed, the body block adds "pass" status behind the "speaker" key, which itself is behind the "audio" key in results_dict. Passing with issues, failure, and cancellation results are done similarly. Sometimes determining the values requires some logic in the body blocks, especially when it comes to canceling, which means that the test cases that have not been executed will not be given a result at all.

The results of the test sets included in the dictionary containing the test case results are deducted based on the results of the test cases. This is done as part of result checking, so the operations have been finished at that point. The deduction logic is explained in Section 3.1.1. The results of PiceaOne test sets and test cases are then verified in the views that are shown in the screenshots in Figures 3.3 and 3.4 on pages 16 and 17. The expected issues and resolutions can always be determined from the result dictionary's results.

Because fMBT can be implemented in connection with various languages and techniques, the comparison tool can vary also. In this study, the comparison of an expected UI element and the shown ones was done using bitmaps. The bitmaps could be easily taken and saved by using the fmbt-scripter tool that was included in the package. Using the fmbt-scripter, a file name from the modeling code could be selected and an area of a screenshot taken from the device selected and connected to the file. On lines 2–6 of Program 5.2, the same actions of checking the UI correctness are done as in Program 4.2 on page 4.2 on lines 3–6. Of course, in the case of checking bitmaps, the way that the UI looks like is verified, whereas checking the existence of elements based on the locator strings does not verify, for example, the location or position of that element rel-

ative to others. The bitmap checking is not absolute either, the required default match of 100% in color channel values has been dropped to 80% to increase stability, so that trivial differences do not cause tests to fail.

5.2.6 Technique evaluation

Implementing model-based test automation for PiceaOne Diagnostics with fMBT was a challenging but very much manageable task. fMBT is a workable tool for implementing MBT for an Android application. Like Robot Framework, fMBT is compatible with a lot of different languages and techniques, any language can be used or invoked via the adapter blocks, including Robot Framework keywords.

The modeling took script abstraction to a higher level in comparison to manually scripted test automation, even higher than keyword-driven testing. The model that was created modeled the functionality of PiceaOne Diagnostics fairly well and to the extent that modifying the model when adjusting to modifications in the SUT was quite easy, i.e., the technique's maintainability was good.

On the downside, as abstraction goes higher, so does difficulty. Modeling requires programming skills and takes a while to grasp. Creating the model was slow, especially at the start. Issues that arose during the modeling were often very complicated, spanning over the whole model and only manifesting in certain parts of it. Even though maintainability felt easy, the possible maintainability problems of a complex system that could be caused by major structural changes were not faced. This was probably a result of the fairly stable architecture of the SUT over the testing period. Still, the ISTQB Syllabus for TAEs gives a hint of such problems by stating that the models also have to be quality assured and maintained [9, p. 36].

The ability to generate test cases automatically based on the model provided the capability to cover significantly more use case scenarios than could be covered when tests for each of these scenarios had to be manually written. The amount of possible execution paths is illustrated by Figure 5.3, which is a visualization of the same part of the model as Figure 5.1 on page 41 (that is also highlighted in Figure 5.2 on page 42), only this time the dictionary containing the results of the test cases is included in the variables affecting the visualization complexity. One can imagine what the visualization of the whole model might look like if each of the PiceaOne test set branches in Figure 5.2 was similarly expanded. The value of the advantage gained from being able to test a large number of scenarios is unclear, and the effects are evaluated in Chapter 6.

The downside of what was described in the previous paragraph, and what Figure 5.3 also illustrates well, is called the test case explosion problem, a common counter-argument to MBT [13, Chapter 1.4]. The test case explosion means that the number of possible combinations grows exponentially based on some variables. As seen in Figure 5.3, with PiceaOne, one such variable is the results dictionary. As the number of PiceaOne test

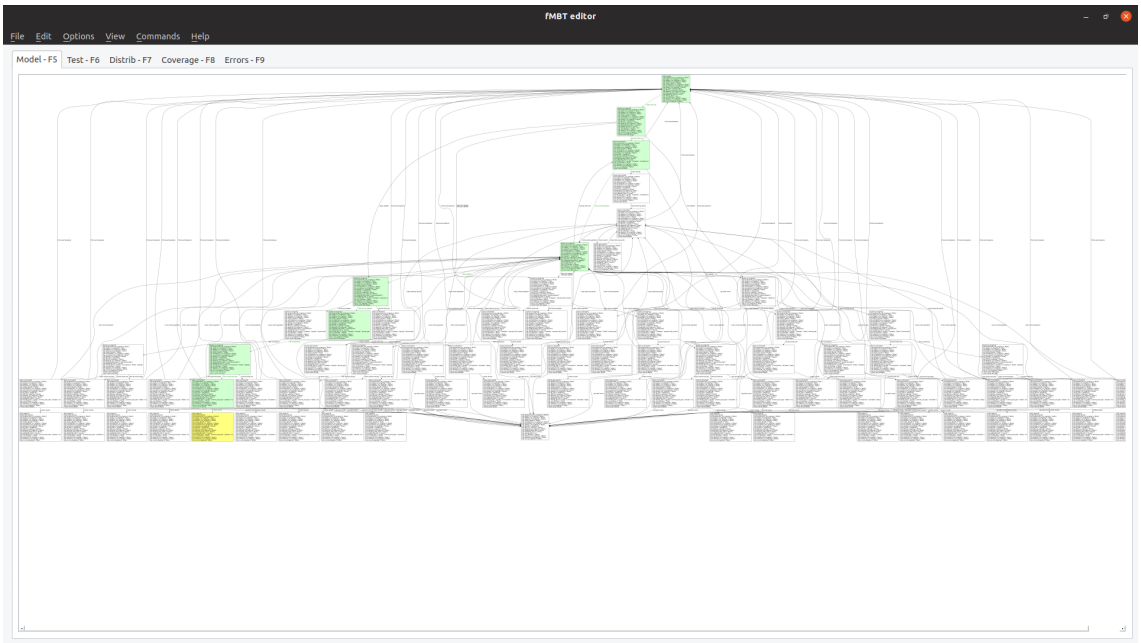

Figure 5.3. *The audio model that is also shown in Figure 5.1 on page 41 complicated with the inclusion of test results in the accounted variables.*

cases increases, so does the number of possible ways to execute them and therefore the amount of unique combinations of results. The problems caused by this issue could partly be mitigated using strategies. For example, a maximum required length could be set for paths, and extremely long or short test runs could be controlled.

A clear advantage of MBT in comparison to both manual testing and manually scripted test automation is that it can be used to run indefinitely long test cases with extreme precision. Moreover, it may select execution paths to be tested that no human would think about, and when the behavior in different situations can be well enough defined in the model, MBT can be used to perform a lot of unexpected events based on random generation. An example of an event like this would be canceling a PiceaOne test case at an arbitrary moment in execution by tapping the device back button. It could be argued that this introduces some creativity, characteristic to exploratory testing, to test automation.

During this study, there were also some problems and advantages with the implemented model-based test automation that had more to do with the technology choices than the actual paradigm of MBT. For example, using bitmaps for verification caused some maintainability issues from minor changes and made using the model for other devices laborious as new screenshots had to be taken. Also, because of bitmaps, when running the tests for a long time continuously, the number of screenshots was large, and therefore memory issues were possible and had to be taken into account. On the other hand, bitmap-based verification could spot errors such as the wrong position or width of an element. Other technical issues were the fact that AAL/Python syntax was more prone to errors caused by minor discrepancies in code and that the fMBT was only usable on Linux.

6 RESULTS AND PERFORMANCE ANALYSIS

The testing in this study was conducted by performing MBT on PiceaOne for three months. The manually scripted test automation, a part of which had been implemented before the start of the study, was run as a point of reference once every workday, using a new version of the application each time a new version was available. A run length was roughly 80 minutes to 100 minutes, varying because of the changes in the SUT, changes in the tests, and changes in timeouts used. After this testing was done, the same version of PiceaOne was used for model-based testing which took an undefined time to finish, at most till it was time to run the manually scripted tests again. This was usually a day, sometimes over the weekend. The idea was to use both implementations to the best of their abilities for each version of the application and see how they performed.

This chapter presents the performance results of the test automation solutions based on the metrics defined in Chapter 2: test step coverage, number, and severity of faults found, and the evaluation of applicability and cost-efficiency over different development phases.

6.1 Problems, inaccuracies, and reliability

There are many problems related to inaccuracies and reliability to take into consideration in a study like this. The inaccuracies are mostly related to estimation, for example, the estimation of the amount of effort put to implementing the techniques. The reliability issues, on the other hand, are mostly related to how often the tests were run successfully and how common were false failure situations.

First off, the fragility, meaning the vulnerability to even minor changes in GUI elements [16] in Android application testing, causes unreliability. The testing environment is not isolated to just the SUT. There are noises, pop-ups, and operating system errors, for instance. This is all the more common for an application like PiceaOne, which is also dependent on external stimuli such as movement, noises, network connections, or ambient light stability. From time to time, entire test runs are compromised, and therefore the output is not a completely stable stream of uncompromising performance results. Some of these reliability issues are also caused by the testing tools rather than the test technique, and that has to be taken into account as the idea is not to compare tools.

Another effect of the tools, the difference in comparison techniques, causes the different bugs to be visible to one technique and invisible to the other. For example, a wrong

width of a UI element goes unnoticed by the element-based checking but is picked up by bitmap-based checking. Finding faults like these should not be expected of a technique that merely verifies the existence of elements. On the other hand, verifying that animations/moving objects are visible is very difficult using bitmap-based checking.

Chance also has a role in the results of this study. Some of the faults found by MBT that were not found by the manually scripted test automation could have been found by the latter also if test cases for certain scenarios had been written. Moreover, the quality of the work and modifications made by the development team is a fairly random factor in what kind of faults happen to be injected into the code.

6.2 Coverage-based analysis

The coverage reached by the MBT was superior to that of the manually scripted test automation. Using the fmbt-editor tool, different coverage statistics could be calculated for the model or certain parts of the model that were created in this study. For example, the necessary amount of test steps could be calculated for reaching a certain coverage of testing for just the audio test set part of the model. With a lookahead depth of 5, which is enough to cover every step in the model, to test each test step at least once, the number of necessary test steps was 25. When required that every permutation of combinations of every 2 test steps are to be tested, the number of cases in the generated script was 108. With 3 being the requirement, the number of test steps was 445. This is the minimum number of steps the test script generator tool can reach that coverage in.

The amount of the same type and size of test steps implemented in the manually scripted test automation was calculated, type, and size meaning that the actions and checks done within a test step were matching those that are considered a step in MBT. For instance, if a test step in the MBT solution included checking the UI elements of the Phone Speaker test case and failing it, the similar actions in the manually scripted automation would be counted as a test step. In total, there were 118 such test steps implemented for the audio test set in the manually scripted test automation. Each of these steps was written by a human, and therefore it is most likely that there is a lot of repetition of certain combinations of test steps. This means that, with the same amount of test steps, the coverage of different permutations is probably not as good as that of the MBT. It would be very difficult and time consuming for a human to try to create tests to test all the permutations of steps that can be made with 3 steps.

The analysis in the previous two paragraphs was only regarding the testing of the audio test set of PiceaOne. If the whole model is included and the required coverage raised even a little bit, it is necessary to generate thousands or tens of thousands of test steps to reach the coverage. fMBT can generate scripts that long if necessary, but it is questionable how much value is added by testing all the permutations of 7 different test steps, for example. As mentioned before, coverage is not necessarily correlated to fault detec-

tion [12, 8]. On the other hand, some studies have found that it has positive effects on fault detection [52, 50]. At least to a certain extent, coverage does give the QA team confidence in the SUT. Knowing that different combinations of tests have been going on for 24 hours without failures helps relieve the QA of testing the "basics" that much.

Another advantage of MBT in terms of coverage is the flexibility it enables. Strategies can be used as mentioned in Section 5.2.3. This means that avoiding the testing of parts of the SUT that are known to be broken or intentionally creating very short or very long test cases is possible. For example, using the strategies it was possible to test every test step of the model once and then run tests without quitting the application for many hours straight. This is an approach similar to what is suggested by Xie and Memon in their paper Studying the Characteristics of a “Good” GUI Test Suite [50].

The manually scripted test automation also had a positive side when it comes to coverage. This had to do with the simplicity of manual scripting. It was much easier to test certain actions in specific parts of a test run with the manually scripted testing. For example, if it was a high priority to test canceling after the first part of the PiceaOne Speaker test case, i.e., in the view that is shown in the middle screenshot of Figure 3.1 on page 14, it would be fairly simple to do this. The TAE would only have to replace tapping the pass or fail button at that point of another test case with a cancel button tap. On the other hand, adding a cancellation like that to the model would require adding another input or a condition. It may not be overly complicated but the more specific the action gets, the more likely it seems to require logic and cause trouble with the model. As mentioned in Section 5.1.1., the model accuracy and abstraction level is something that has to be made compromises with.

Based on the fact that MBT offers better coverage in general and the ability to control and focus the desired coverage on a certain area or to a certain combination of actions, it seems to perform very well in comparison to the manually scripted test automation. Besides, MBT adds possibilities to testing that are very hard or laborious to reach even with manual, human-performed testing: the ability run and keep rerunning an operation for extremely long periods and the ability to accurately cover a large set of permutations that a human would have a hard time keeping track of. On the downside, covering very specific sequences of events may complicate the model too much to make it worth testing those.

6.3 Analysis based on found faults

The MBT solution found 12 faults in the SUT, while the manually scripted solution found 3. All of the faults found by the manually scripted solution were also found with the MBT solution. Not all of the faults found by MBT can be attributed to the paradigm of MBT, but in 8 of the cases, it could be argued that the model-based technique may have contributed to detecting the faults.

The faults found were categorized by their severity from blocker (0) to low-low (9). The requirement for a release is that no blocker (0) or high (1-3) category faults are known to be in the code. The severities of the faults found by the two test automation solutions were the following: MBT found 2 high-medium (2), 2 medium-high (4), 6 medium-medium (5), 1 medium-low (6) and 1 low-medium (8) severity faults. Of those, the manually scripted test automation found 1 high-medium and 2 medium-medium faults. The average severity was 4 for manually scripted automation and 4.75 for MBT. In other words, there was no significant difference in the severity of the faults found by the techniques. The low severity of two of the faults found by MBT is because of their extremely rare occurrence.

The faults found by both of the solutions included a fault that caused the accessory connector test case of PiceaOne Diagnostics to falsely pass, the failure of PiceaOne to report an issue for failed accessory connector test case and a situation in which PiceaOne was showing an issue of a failed embedded SIM (eSIM) test case for a device that does not support eSIM. The first one of these was considered high-medium in severity, the other two medium-medium.

Faults that were picked up by MBT but should not be taken into account in the performance comparison included a fault in which almost every UI instruction element (the rectangles with blue background in Figure 3.1 on page 14, for example) of PiceaOne was slightly wrongly constrained or scaled. Most of these issues were not noticeable to the human eye but in some cases the text of the instruction element did not fit in the area meant for it. Testing buttons was left out of the manually scripted test automation as no suitable solution for this was found.

There were two faults found by MBT that were the result of testing Android buttons. Even though MBT should not be compared to manually scripted testing in this instance, it should be noted that finding these two faults could be attributed to MBT automated test generation thoroughness.

Several of the faults found can be attributed to the MBT technique's superior coverage. For example, there were three medium-medium severity faults that occurred by tapping the cancel button at a certain point of PiceaOne test operation. These did not occur every time and were never found by the manually scripted test automation even though canceling in a similar way was included in the test scripts. These issues were the failure of PiceaOne to cancel or start sensors tests by tapping cancel or the screen which is supposed to start the tests, the failure to show correct "canceled" result after successful sensors test set cancellation and a situation in which PiceaOne connectors test set was never finished, causing the operation never to finish. The issues with canceling sensors, connectors and accessories test sets were numerous and some of them were caused by the same fault. MBT was useful in highlighting the numerous occasions in which the issues occurred which provided useful information regarding the fault to the developers.

Other faults that were picked up by MBT but not by the manually scripted automation include a failure to stop recording in front camera video test even though stop recording

button was tapped, showing a wrong piece of instructions in connectivity test set and a situation in which PiceaOne ended up in the Device Details view after finishing operation instead of the results view shown in the leftmost screenshot in Figure 3.3 on page 16. Another trivial issue was the wrong order of presenting instructions in the connectivity test set. These issues could have been picked up by test automation if it had better coverage.

It should be noted here that the Robot Framework test automation was already partly being used when this study was started, and some of the issues it had found had already been fixed. Still, the manually scripted test automation was expanded for this study, and the MBT seemed to find every one of the new issues found by the manually scripted test automation and some more. In short, MBT found more faults than the manually scripted test automation, even when taking into account just the faults that could have also been found by the manually scripted test automation. The severity of the faults found was roughly the same on average, slightly in favor of the manually scripted test automation.

The reason for the low severity of some of the faults found by MBT was their rarity. Two of the faults found by MBT could not be reproduced by manual testing despite the hard effort, and a third one could be reproduced from time to time. The fact that the faults do not easily reproduce in human use of the SUT lowers their severity, but it also goes to show the potential of MBT to find faults that would be extremely difficult for a human to find.

6.4 Secondary metrics analysis

The cost-efficiency of manually scripted test automation in terms of the type of test step coverage discussed earlier in this chapter seems to be fairly linear. Getting started with the test cases was fairly fast and the keyword-driven approach provided some leverage after the implementation of user-defined keywords. However, after a basic coverage was reached, the addition of new test cases had a less and less relative effect. The cost-efficiency in different phases of implementation was different for MBT. Creating a model was slow at the start and several steps back had to be taken to find a suitable way to do it. For example, making the model check Diagnostics tests' results after finishing a PiceaOne test set required the temporary exclusion of test sets that were not yet fully finished or for which result checking was not fully finished. After the model got closer to being finished, the potential of MBT was manifested as an exponential growth of coverage in relation to the effort put to developing it.

The ease of implementation and the intuitiveness of manually scripted test automation made it a winner in terms of applicability. Scripts could be easily created for multiple devices by separating parts of the scripts that were characteristic for certain devices only into separate files. Of course, the element-based comparison of what is shown on the UI was favorable for multiple devices, but keyword-driven scripting as a technique was also

useful for multiple devices. For example, painting the screen for devices with different screen sizes could simply be defined as a keyword, the definition for which could be different for different devices. Another major advantage of the Robot Framework was that the same system worked for iOS application testing as well.

As a tool, the Robot framework was a more flexible solution when it comes to platform and device variability, but this was also a major weakness for it in the form of a multitude of necessary dependencies. These dependencies caused a lot of reliability issues, especially in the early parts of the study. For example, issues with communication with the Android system and different types of timeout errors were common.

7 CONCLUSION

The conclusions drawn from this study are mainly in line with the theory presented in Chapters 4–5. The generally acknowledged strengths and weaknesses of MBT in comparison to manually scripted test automation are present in the Android application environment. MBT achieves better coverage and fault detection rates after a critical amount of effort is put to it.

Implementing model-based testing requires some programming skills and effort, as well as an understanding of the SUT. When implementing MBT, attention should be paid on selecting the type of model and notation because using an unfit solution may be costly. Also, a model should not be too detailed, nor too abstract. In comparison to MBT, implementing test automation by manual scripting is straightforward and intuitive. It does not require a deep understanding of the SUT, and it should not include the chances of making costly mistakes through big decisions.

The performance of MBT, in terms of use case coverage, is modest at the early stages of development. This is the result of the fact that modeling at early stages tends to be related to abstract parts of the SUT that may not form a testable entirety until a larger part of the model is complete. Until that point, manually scripted performs relatively well as adding simple test cases merely requires scripting a linear execution path. However, once the model starts resembling the SUT well enough, the coverage achieved by it should grow exponentially and quickly overtake that of manually scripted test automation.

The superior performance of MBT in terms of coverage in the long run results in it being superior also in finding faults in the long run. This is not only caused by the fact that writing scripted tests with comprehensive use case coverage is hard work but also a result of the ability of MBT to effectively find unique use case scenarios, including ones that humans may never think about. Other distinguishable advantages of MBT are that it can perform extremely long test cases and that it can be modeled to use pseudorandom inputs and yet know what results to expect.

Basing on the points made in the three paragraphs above, MBT is a great approach for GUI test automation when there is a need to test an Android application comprehensively. It is beneficial, especially when done well, and when the SUT is complex enough to require complicated testing sequences. MBT is a preferable test automation option when the excellent quality of the software is a high priority, testing for memory leaks is essential, and when thorough testing is required in general.

Model-based testing is not the best option for testing small or otherwise simple applications, or if the goal of test automation is just to automate some basic testing. The maintenance of model-based testing when it comes to reacting to small changes in the SUT is fairly easy, but it may require more effort when facing larger architectural changes. When committing to MBT, one should prepare to put a lot of effort into it and not expect great results early.

REFERENCES

[1] *Agile Alliance Glossary. Agile Alliance*. URL: https://www.agilealliance.org/glossary/bdd (visited on 08/05/2019).

[2] E. Alégroth, R. Feldt, and P. Kolstrom. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. English. In: *Information and Software Technology* 73 (2016), 66. ISSN: 0950-5849. URL: https://arxiv.org/abs/1602.01226 (visited on 09/21/2019).

[3] E. Alégroth, R. Feldt, and L. Ryrholm. Visual GUI testing in practice: challenges, problems and limitations. English. In: *Empirical Software Engineering* 20.3 (2015), 694–744. ISSN: 1382-3256. DOI: 10.1007/s10664-013-9293-5.

[4] *Appium Home Page. appium*. URL: http://appium.io/ (visited on 08/20/2019).

[5] *Appiumlibrary Github*. URL: https://github.com/serhatbolsu/robotframework-appiumlibrary (visited on 08/20/2019).

[6] Y. L. Arnatovich and L. Wang. A Systematic Literature Review of Automated Techniques for Functional GUI Testing of Mobile Applications. English. In: (2018). URL: https://arxiv.org/abs/1812.11470 (visited on 09/21/2019).

[7] A. Axelrod. *Complete Guide to Test Automation : Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. English. Berkeley, CA: Apress L. P, 2018. ISBN: 9781484238318.

[8] D.-H. Bae, G. Bae, and G. Rothermel. Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study. English. In: *The Journal of Systems and Software* 97 (2014), 15–46. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.06.039.

[9] B. Bakker, G. Bath, A. Born, M. Fewster, J. Haukinen, J. McKay, A. Pollner, R. Popescu, and I. Schieferdecker. *Certified Tester Advanced Level Syllabus - Test Automation Engineer*. English. ISTQB, 2016. URL: https://www.istqb.org/downloads/send/48-advanced-level-test-automation-engineer-documents/201-advanced-test-automation-engineer-syllabus-ga-2016.html (visited on 07/05/2019).

[10] L. Bandeira, P. Donegan, C. Maia, C. Matos, and P. da Cunha. *Automated Software Testing*. English. 2007.

[11] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. English. In: *IEEE Transactions on Software Engineering* 41.5 (2015), 507–525. URL: https://ieeexplore.ieee.org/document/6963470 (visited on 09/21/2019).

[12] V. R. Basili and R. W. Selby. Comparing the Effectiveness of Software Testing Strategies. English. In: *IEEE Transactions on Software Engineering* SE-13.12 (1987), 1278–1296. ISSN: 0098-5589. DOI: 10.1109/TSE.1987.232881.

[13] R. V. Binder, B. Legeard, G. Bazzana, and A. Kramer. *Model-based testing essentials: guide to the ISTQB Certified Model-Based Tester foundation level*. English. Wiley-Blackwell, 2016. ISBN: 1119130018.

[14] S. Bisht. *Robot framework test automation*. English. 1st ed. Birmingham: Packt Publishing, 2013. ISBN: 1783283041.

[15] R. Black, J. McKay, G. Bath, D. Friedenberg, H. B. D., K. Onishi, M. Smith, G. Thompson, and T. Yumoto. *Certified Tester Advanced Level Syllabus - test manager*. English. ISTQB, 2012. URL: https://www.istqb.org/downloads/send/10-advanced-level-syllabus-2012/54-advanced-level-syllabus-2012-test-manager.html (visited on 06/27/2019).

[16] R. Coppola, M. Morisio, and M. Torchiano. Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility. English. In: ACM, 2017, 22–32. ISBN: 9781-450353052. URL: https://arxiv.org/pdf/1711.03565.pdf (visited on 09/21/2019).

[17] R. D. Craig and S. P. Jaskiel. *Systematic software testing*. English. Boston: Artech House, 2002, 536. ISBN: 9781580537926.

[18] L. Crispin and J. Gregory. *Agile testing: a practical guide for testers and agile teams*. English. 1st ed. Upper Saddle River, NJ: Addison-Wesley, 2008. ISBN: 0321534468.

[19] S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson, and D. Sundmark. A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques. English. In: IEEE, 2006, 159–170. ISBN: 9780-769526720. DOI: 10.1109/TAIC-PART.2006.1.

[20] E. Engström, P. Runeson, M. Skoglund, D. of Computer Science, E. the Linköping-Lund initiative on IT, mobile communication, L. University, I. för datavetenskap, and L. universitet. A systematic review on regression test selection techniques. English. In: *Information and Software Technology* 52.1 (2010), 14–30. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.07.001.

[21] *fMBT android help. fMBT Android help*. URL: https://01.org/fmbt/blogs/ask/2013/fmbt-android-ui-testing (visited on 08/29/2019).

[22] *fMBT homepage. fMBT*. URL: https://01.org/fmbt (visited on 07/31/2019).

[23] *fMBT tutorial. fMBT Tutorial*. URL: https://github.com/intel/fMBT/wiki/Tutorial (visited on 08/29/2019).

[24] *Github AAL Syntax. AAL*. URL: https://github.com/intel/fMBT/blob/master/doc/aal_python.txt#L99 (visited on 08/28/2019).

[25] A. M. J. Hass. *Guide to advanced software testing*. English. Boston: Artech House, 2008. ISBN: 1596932864.

[26] B. Hetzel. *Making Software Measurement Work: Building an Effective Measurement Program*. New York, NY, USA: John Wiley & Sons, Inc., 1993. ISBN: 0894354655.

[27] W. S. Humphrey. *Introduction to the Personal Software Process(SM)*. English. First edition. Addison-Wesley Professional, 1996. ISBN: 0201548097.

[28] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/metric (visited on 09/05/2019).

[29] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/measurement (visited on 09/05/2019).

[30] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/inspection (visited on 09/04/2019).

[31] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/test%20automation (visited on 07/22/2019).

[32] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/test%20procedure (visited on 09/04/2019).

[33] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/test%20oracle (visited on 07/22/2019).

[34] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/model-based%20testing (visited on 07/23/2019).

[35] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/scripted%20testing (visited on 07/31/2019).

[36] *ISTQB Glossary. ISTQB*. URL: https://glossary.istqb.org/en/search/model-based%20%20test%20strategy (visited on 07/23/2019).

[37] P. C. Jorgensen. *Software Testing: A Craftsman's Approach, Fourth Edition*. English; Portuguese. Auerbach Publications, 2013. ISBN: 9781439889503.

[38] P. C. Jorgensen. *The Craft of Model-Based Testing*. English. 1st ed. London: Auerbach Publications, 2017. ISBN: 1498712290. DOI: 10.1201/9781315204970.

[39] R. Kirner and W. Haas. Optimizing compilation with preservation of structural code coverage metrics to support software testing: PRESERVATION OF STRUCTURAL CODE COVERAGE METRICS. English. In: *Software Testing, Verification and Reliability* 24.3 (2014), 184–218. DOI: 10.1002/stvr.1485.

[40] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the Test Automation Culture of App Developers. English. In: IEEE, 2015, 1–10. ISBN: 2159-4848. DOI: 10.1109/ICST.2015.7102609.

[41] L. Lazic and N. Mastorakis. Cost Effective Software Test Metrics. In: *W. Trans. on Comp.* 7.6 (June 2008), 599–619. ISSN: 1109-2750. URL: http://dl.acm.org/citation.cfm?id=1458369.1458372.

[42] M. Linares-Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do Developers Test Android Applications? English. In: IEEE, 2017, 613–622. DOI: 10.1109/ICSME.2017.47.

[43] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. English. In: ACM, 2008, 131–142. ISBN: 1605-580503. DOI: 10.1145/1390630.1390648.

[44] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. English. In: ACM, 2010, 1–4. ISBN: 1450-30138X. DOI: 10.1145/1868048.1868049.

[45] S. Morasca and S. Serra-Capizzano. On the analytical comparison of testing techniques. English. In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), 154. ISSN: 0163-5948. DOI: 10.1145/1013886.1007533.

[46] *Oxford Advanced American Dictionary, definitions for performance. Oxford Learner's Dictionary*. URL: https://www.oxfordlearnersdictionaries.com/definition/american_english/performance (visited on 09/05/2019).

[47] *Robot Framework Home Page. robot framework*. URL: https://robotframework.org (visited on 08/20/2019).

[48] J. Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. English. 1. Aufl.; 1. New York, NY: Wiley, 2005. ISBN: 9780471713456. DOI: 10.1002/0471722324.

[49] M. Utting and B. Legeard. *Practical Model-Based Testing*. English. 1st ed. Morgan Kaufmann, 2010. ISBN: 0123725011.

[50] Q. Xie and A. M. Memon. Studying the Characteristics of a "Good" GUI Test Suite. English. In: IEEE, 2006, 159–168. ISBN: 1071-9458. DOI: 10.1109/ISSRE.2006.45.

[51] Q. Xie and A. Memon. Designing and comparing automated test oracles for GUI-based software applications. English. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16.1 (2007), es. ISSN: 1049-331X. DOI: 10.1145/1189748.1189752.

[52] X. Yuan, M. B. Cohen, and A. M. Memon. GUI Interaction Testing: Incorporating Event Context. English. In: *IEEE Transactions on Software Engineering* 37.4 (2011), 559–574. DOI: 10.1109/TSE.2010.50.